The Dissertation Committee for David Lawrence Rager
certifies that this is the approved version of the following dissertation:

Parallelizing an Interactive Theorem Prover: Functional Programming and Proofs with ACL2

Committee:

Warren A. Hunt, Jr., Supervisor

James C Browne

Matt Kaufmann

J Strother Moore

Jun Sawada

Emmett Witchel

Parallelizing an Interactive Theorem Prover: Functional Programming and Proofs with ACL2

by

David Lawrence Rager, B.B.A.; M.A.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2012

Dedicated to my family.

Acknowledgments

Finishing a dissertation is a non-trivial task, and I could have not done it without the support of many people. I thank Warren Hunt for his support and encouragement throughout graduate school, and for supervising this work. I also thank him for his focus on improving the performance of ACL2, as without that, the resources necessary to parallelize such a large piece of software may not have been available. I thank Matt Kaufmann for spending numerous hours helping me to obtain the knowledge necessary to implement ACL2(p), developing ACL2 modifications to support this work, and integrating and helping to improve the ACL2(p) code; and also for his general support and encouragement. I thank J Strother Moore for being an inspiration to us all and providing feedback when appropriate. I thank James Browne, Jun Sawada, and Emmett Witchel for making the time to serve by reviewing and providing advice concerning the content and direction of this dissertation. I thank my parents, Diane and Brent Rager, for being such outstanding examples to follow, and I thank my brother Jonathan and sister Julia for being a positive influence in my life. I thank Sharon Kuhn for reading the dissertation and providing feedback. I thank Jared Davis for always having an open ear and his interest in my work and life in general. I thank Nathan Wetzler for providing feedback on various ideas I have had. I thank Robert Krug and Shilpi Goel for being early adopters of ACL2(p) and providing feedback on its

interactive usefulness. I thank Gary Byers, Martin Simmons, and the SBCL community for providing Lisp support and feedback on our implementation. I thank Sung Jun Lim for verifying some of my results and relevant discussions.

Parallelizing an Interactive Theorem Prover: Functional Programming and Proofs with ACL2

Publication No. _____

David Lawrence Rager, Ph.D.
The University of Texas at Austin, 2012

Supervisor: Warren A. Hunt, Jr.

Multi-core systems have become commonplace, however, theorem provers often do not take advantage of the additional computing resources in an interactive setting. This research explores automatically using these additional resources to lessen the delay between when users submit conjectures to the theorem prover and when they receive feedback from the prover that is useful in discovering how to successfully complete the proof of a particular theorem.

This research contributes mechanisms that permit applicative programs to execute in parallel while simultaneously preparing these programs for verification by a semi-automatic reasoning system. It also contributes a parallel version of an automated theorem prover, with management of user interaction issues, such as output and how inherently single-threaded, user-level proof features can be configured for use with parallel computation. Finally, this

dissertation investigates the types of proofs that are amenable to parallel execution. This investigation yields the result that almost all proof attempts that require a non-trivial amount of time can benefit from parallel execution. Proof attempts executed in parallel almost always provide the aforementioned feedback sooner than if they executed serially, and their execution time is often significantly reduced.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

ACL2 is a theorem prover for first-order logic with induction based on an applicative subset of Common Lisp. It has been used in some of the largest, industrial formal verification efforts [8, 54, 16, 48, 51, 49, 47, 50]. As multi-core systems become commonplace, theorem prover users would like to take advantage of the additional available hardware resources [24, Section 4.5]. Since the ACL2 theorem prover is primarily written in its own functional language, and since one can introduce parallel execution into functional languages with fewer difficulties than typically encountered when parallelizing an imperative program (see Section 1.4 in *An overview of actor languages* [2]), parallelizing the execution of ACL2 is a more obtainable goal than parallelizing many other interactive programs. *By using parallel execution in ACL2's main proof process, we decrease the delay between when users submit conjectures to the prover and when they receive useful feedback from the prover concerning how to guide the theorem prover to a successful proof.* Since the people who are trained in theorem proving tend to be both expensive and rare, this reduction in debugging time for failed proofs is beneficial to those developing the proofs, the institutions that employ them, and the research community in general. Finally, our approach (see Section 1.2 for an explanation of this approach)

improves these users' interactive experience without compromising soundness.

1.1 Contributions of this Work

This dissertation explores three main areas:

- The development of mechanisms to permit applicative programs to execute in parallel, while simultaneously preparing these programs for verification by a semi-automatic reasoning system (explained in Section 1.1.1),

- The development of a parallel version of an automated theorem prover, with management of user interaction issues like providing coherent output and how inherently single-threaded user-level proof features can be configured for use with parallel computation (explained in Section 1.1.2), and

- An investigation into the types of proofs that are amenable to parallel execution (explained in Section 1.1.3). This investigation yields the result that almost all proof attempts that require a non-trivial amount of time can benefit from parallel execution; under parallel execution, proof attempts will almost always provide the aforementioned feedback sooner than if they executed serially, and their execution time can often be significantly reduced.

1.1.1 Parallelism Primitives

We allow users to introduce parallel execution into an ACL2 program by:

- Introducing low-level multi-threading primitives for interfacing with the underlying runtime system. We then build higher-level parallelism abstractions on top of the low-level primitives. These abstractions empower users to identify computations that can be parallelized while still preserving the functional model of their code.
- Defining these abstractions such that proofs about the serial version of their program and proofs about the parallel version of their program are the same. This allows users to benefit from parallel execution without spending any of their time reworking the proofs about their program.
- Introducing these abstractions in a manner that does not alter the ACL2 logic. Preserving the logic in this way means we do not risk adding a feature to the logic that could compromise the soundness of ACL2.

This is one of the first parallelized extensions of a functional programming language that has an associated reasoning system (see Section 2.4 for discussion of another modern system with related capabilities). This extension improves ACL2's rapid-prototyping capability, because, with the use of our primitives, (1) users can write purely functional code that executes in parallel and (2) they are able to formally verify properties of their parallel code without any additional proof overhead.

1.1.2 Parallelized Theorem Prover

We have developed a parallel version of the ACL2 theorem proving system by (1) reducing the number of serial dependencies in the main theorem proving process so that a greater portion of it can be executed in parallel and (2) using the mechanisms mentioned in the previous section to introduce parallel execution into the main proof process. We have accomplished this by:

- Identifying and removing sequential dependencies from the main proof process.
- Presenting output such that coherent feedback is given sooner to help users more quickly investigate failed proof attempts. This is non-trivial because, when the prover is trying to solve several goals at once, explaining what is happening becomes much more difficult. We developed a mechanism that very quickly presents the subgoals that fail to prove, so users can begin debugging their proof attempts sooner.
- Continuing to support the ways that users interact with ACL2, even though there is now parallel execution in its proof process. This includes the reworking of inherently sequential hints that users may provide to the theorem proving process.
- Providing a runtime mechanism for toggling between parallel and serial modes of execution. This allows users to make the transition to the parallel mode more smoothly. It also permits users to swap back to

serial execution when they need a feature not supported under parallel execution.

- Investigating heuristics for determining when to parallelize the proofs of subgoals, and providing mechanisms to users for configuring the use of these heuristics.

Our modification of ACL2 that supports performing proofs in parallel is called ACL2(p), is distributed with the main distribution of ACL2, and is available to users as a compile-time flag. ACL2(p) illustrates how an interactive theorem proving system can be programmed to execute in parallel, even when there are multiple serial dependencies between the various procedures that implement the theorem proving process. ACL2(p) provides an improved interactive user experience by facilitating faster model development and verification.

1.1.3 Proofs Amenable to Parallel Execution

We have investigated and classified the types of proofs that are amenable to parallel execution at the subgoal level. One might expect that in an ideal world, all proofs would experience linear speedup with respect to the number of CPU cores in the system, and theorem prover users would immediately receive the feedback most relevant to debugging their proofs. However, in practice, this is not the case. For example, proofs about code with several conditionals can generate many independent subgoals (generating subgoals in

this way is typically called a "case-split"), each of which can be proved in parallel. However, lengthy proofs that do not contain such case-splits also exist. We categorize proofs with respect to the improvement in overall proof time and the improvement in how long it takes users to receive feedback that leads to correcting their proof attempt.

We determine the usefulness of subgoal-level parallelism across a variety of tests, taken from the ACL2 regression suite. These tests are written by many users, each with their own style; the tests also cover a diverse set of subjects, including arithmetic, hardware verification, graph theory, security, etc. Although many of these finished products require a relatively short amount of processing time, many of them still require a non-trivial amount of time to complete and will demonstrate the capability of providing feedback to users more quickly.

1.2 Goal of ACL2(p): Reduce Interactive User Time

The main goal of ACL2(p) is to reduce the time it takes users to receive useful feedback in an interactive setting. Since any theorem developed in an interactive setting eventually becomes part of a suite of theorems that are certified non-interactively, we can rely on the soundness of the version of ACL2 used in the non-interactive sessions – specifically the version of ACL2 that does not allow parallel execution. As long as users use the non-parallel version of ACL2 to certify their proofs non-interactively (such batch non-interactive certification is the de facto standard), any proof that takes advantage of an

underlying flaw in the implementation of ACL2(p) will fail the final test. With this failsafe in place, as developers of ACL2(p), we are able to take some risks and think through some of the necessary changes less carefully than if we were attempting to guarantee 100% soundness. This risk-reward tradeoff allows us to finish a useful parallel version of ACL2 in a reasonable time-frame. That said, we do not expect users to encounter soundness bugs in practice.

1.3 Key Results

As described in Section 1.1.3, this dissertation surveys the ACL2 regression suite to determine the usefulness of parallelizing a theorem prover at the subgoal level. Our key results include performance statistics for these proofs on three different machines. Most of the proofs we analyze were not created with parallel execution in mind. These include proofs about the Java Virtual Machine; for example, theorem *[2b]* experiences a speedup of 13.44x on a twenty core machine. In an effort to explore the potential speedup that can occur from parallel execution, we designed a proof, named *ideal-40-way*, to achieve close to a 20x speedup on a twenty core machine. And indeed, this proof achieves a speedup of 18.93x on our twenty-core test machine. For the top 200 longest proofs in the regression suite, the average speedup on our twenty core test machine is 3.69x (with `resource-based` parallelism, which is explained in Section 6.1; the average speedup with `full` parallelism, explained in the same section, is 3.66x). This reduction in the total amount of time required by a proof attempt is one way that users obtain feedback sooner than

if the proof were to execute serially.

The second way that we provide feedback sooner occurs as a by-product of concurrently executing the proofs of subgoals. In the past it could take a large amount of time to receive feedback concerning a subgoal that could not be proven with the lemmas already provided to the theorem prover. Now that we have implemented parallel execution within the proof process, as discussed in Section 7.2.3, most proof attempts can now parallelize their execution of subgoals early enough such that this feedback is typically available to users much sooner than it used to be. Provided we used more than one thread to parallelize the execution of the waterfall, this second type of early feedback could occur even on a machine with only a single CPU core.

1.4 Organization of the Dissertation

This dissertation begins by giving a brief background of parallelism in Lisp and other functional and procedural languages. We then explore parallelism within the theorem proving community, including work done on Isabelle/HOL and other work done to parallelize ACL2. After discussing this related work, we introduce, in Chapter 3, our target application: ACL2 and its main proof process. In Chapter 4, we describe the parallelism primitives that we created in order to parallelize the execution of ACL2. Within this chapter we assess the efficiency of these primitives with toy definitions of the Fibonacci function.

Prior to delving into the results of using our parallelism primitives

within the theorem prover, it is necessary to understand the potential performance of the underlying runtime systems. Therefore, in Chapter 5, we introduce our three test machines and discuss their performance on simple Lisp programs.

In Chapter 6, we describe the mechanisms available to users for managing parallel execution. For example, we provide a method to dynamically enable and disable different configurations for parallelizing the main proof process. We also explain in this chapter which of these configurations are useful and which exist only for experimental purposes. Furthermore, because, in the non-parallel version of ACL2, users can modify the global program state from within the proof process, we describe how we disallow that capability. The remaining portion of Chapter 6 describes our mechanism for configuring the amount and type of output that users receive while executing the proof process in parallel.

We then, in Chapter 7, categorize proofs based on how they benefit from parallel execution. We present case studies for each of these categories, where we include traces that show when subgoals are able to start and finish execution. At this point in the dissertation, the reader is equipped to assess our most significant findings: the proof performance improvements presented in Section 7.3.

Parallelizing the execution of ACL2's main proof process could be useless without the support of an efficient parallel execution engine. Chapter 8 discusses some of the design decisions, heuristics, optimizations, and debugging

capabilities that our work includes. After discussing these implementation-specific details, Chapter 9 outlines the procedure that we followed to implement many of the changes to the ACL2 system discussed throughout this dissertation. Finally, we conclude and describe future work, both for the implementation of ACL2(p) and in the area of automated interactive theorem proving in general.

Chapter 2

Related Work

In this chapter we summarize the following bodies of related work: parallelism in Lisp (in Section 2.1), parallelism in other functional languages (in Section 2.2), parallelism in procedural languages (in Section 2.3), and parallelism in theorem provers (in Section 2.4).

2.1 Parallelism in Lisp

There is a rich history of work on parallelizing implementations of Lisp, such as Multilisp. Multilisp was created in the early 1980s as an extended version of Scheme [14]. It implemented the *future* operator, which is often defined as a promise for a form's evaluation result (see Section 4 in *New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools* [15]). In this dissertation, we provide an extension to ACL2 that includes futures, which is discussed later in Section 4.2.1, but we do not integrate the future operator into the ACL2 logic.

Other parallel implementations of Lisp include variants such as Parallel Lisp [15], a Queue-based Multi-processing Lisp [12], and projects described in Yuen's book *Parallel Lisp Systems* [65]. Our approach is different from

previous approaches, in that we embed the logical abstractions described in Section 4.2.2 into the language and logic of a theorem proving system.

As stated in the introduction, one contribution of our work is the further development of a multi-threading library for Lisp systems, specifically for Clozure Common Lisp [9], Steel Bank Common Lisp [55], and LispWorks [30]. The Bordeaux Threads project is an example of a library that already attempts to unify the multi-threading interfaces of different Lisps [7]. However, since the Bordeaux-threads project makes different decisions than we wish to make, we continue the development of our own multi-threading interface.

One of the attributes of Common Lisp that lends itself to partitioning work into pieces and easily passing those pieces to other threads is the *closure*. Closures allow us to easily pass around references to objects and save data to shared memory without needing explicit knowledge of the entire namespace in each thread. Another feature of Lisp that we use is that as long as a variable is defined globally and declared `special`, it is automatically accessible from all threads. And, in our Lisp implementation targets, it is possible to rebind a thread-local version of a global variable with a simple `let` binding.

Another reason that ACL2 (as a Lisp subset) is particularly well-suited for parallelism is that the functional programming paradigm lends itself naturally to parallel execution. As an example, one can imagine spawning a thread for each function in the computation `f` o `g` o `h` (which reads "`f` compose `g` compose `h`"). In this example, partial results from `h`'s execution could be passed to `g` before `h` finishes computing. And likewise, partial results from

`g`'s execution could be passed to `f` before `g` and `h` finish computing. This is function-level parallelism, and we do not use this type of parallelism in this work. Another option would be to give the processing of each element of a list to a thread. So if one has a list of length n, one obtains n-way parallelism. This is an example of data-level parallelism and is the type of parallelism that we implement inside ACL2's main proof process. Lastly, a refinement of data-level parallelism involves a more hierarchical approach. Instead of partitioning the list one element at a time, the list can be partitioned into sections, similar to a mergesort, and different sections can be given to different threads. Our implementation of this hierarchical approach is explained in sections 3.1.2 and 8.3.2.3.

2.2 Parallelism in Other Functional Languages

A second body of work concerns other functional and procedural parallelism implementations. As an example, Haskell is a widely-used, functional programming language that has parallelism variants. Ordinary single-threaded Haskell programs do not benefit from enabling SMP parallelism. Users must expose parallelism to the compiler by either explicitly creating threads or using a "`par`" operator. `Par` has the type signature `par :: a -> b -> b` and, when used in an expression such as `(x ‘par‘ y)`, executes `x` in a separate thread and returns the value of `y`. This can be useful because the expression that `y` represents can reference the value computed by evaluating `x` (when this occurs, the execution of `y` will block until `x` has finished executing). For

further details, see Section 7.22 in the *Glasgow Haskell Compilation System User's Guide* [19].

Orc is another functional language with parallelism included as a first-class feature. In the Orc language, there are four combinators: the parallel combinator "`|`", the sequential combinator "`>x>`", the pruning combinator "`<x<`", and the less well-known *otherwise* combinator "`;`". The parallel combinator is used to spawn threads to evaluate expressions in parallel, and the other combinators help manage the flow of results from those parallel evaluations. Further details can be found in *The Orc Programming Language* [27].

LabVIEW is a graphical programming language first released in 1986, created by National Instruments [10]. LabVIEW implements a dataflow paradigm that looks a lot like a flowchart or circuit design diagram – there are components and wires that connect each component. This type of language feels a lot like a functional language, because data flows through the wires of the design, provides input to the components, and then the components provide output values. This design naturally lends itself to parallel execution, because each node could, in principle, be simulated in a separate thread. Furthermore, this parallel simulation can be performed in a manner that is transparent to the user, allowing the programmer to take advantage of the extra CPU cores without doing any extra work. We mention this work because of its use throughout the electrical engineering industry today, and because it is a nice example of how parallel execution has been provided in a way that does not require user annotation of programs.

Concurrent ML (CML) [52, 53] is an extension of Standard ML [36] that provides programmers with primitives that allow ML programs to execute in parallel. These mechanisms include threads, channels (which provide a means for threads to communicate with one another), and many other primitives. CML has been used in several applications, including a multi-threaded GUI toolkit [13], a distributed tuple-space implementation [53], and a system for implementing partitioned applications in a distributed setting [64]. There are also other parallel extensions to ML, including BSML [31], MSPML [32], and Manticore [11].

2.3 Parallelism in Procedural Languages

Our third body of related work involves the multi-threading paradigms of imperative languages like C++ and Java, which are well known for their focus on synchronization primitives like condition variables, locks, shared memory, and threads. One parallelism extension for C that operates in a way that requires less use of such synchronization primitives from the programmer is Cilk [37]. Cilk is a C extension that provides parallel execution with a very fine level of granularity. Cilk attempts to keep each CPU core busy, and once a thread finishes executing its current workload, it "grabs" another piece of computation by "stealing" some of another thread's remaining work from its stack frame. While we could modify some Lisp implementations to allow direct access to stack frames, the Lisp implementations we use already provide the multi-threading primitives sufficient for executing the main proof process in

parallel (we elaborate upon the issue of granularity in Section 8.3.1). So, while Cilk's techniques are not directly related to this work, the ability to provide parallelism for such a low level of granularity is noteworthy.

Several frameworks enable distributing computation across networks in C++, including Unified Parallel C (UPC) [28] and Message Passing Interface (MPI) [4]. Since our primary goal is to improve the ACL2 theorem prover user's interactive experience, and gains in this area can be made just by using SMP parallelism, we leave the distribution of ACL2's proof process across a network as future work.

2.4 Parallelism in Theorem Provers

The fourth and final body of related work that we discuss is parallelism in theorem provers. We define a parallelized theorem prover as a theorem prover that conducts portions of proof attempts in parallel with one another. Kapur and Vandevoorde developed DLP [23], a distributed version of the Larch Prover, as a framework for interactive theorem proving that takes advantage of *or-parallelism*. Like ACL2, DLP is a rewrite-based theorem prover with many opportunities for the parallelization of subgoal proofs. Kapur and Vandevoorde recognize both the potential for speedup and the need to support user interaction. DLP provides a primitive named *spec* (short for speculate), which applies a user-specified proof strategy to a conjecture or subgoal. These strategies include case splitting and induction. ACL2 provides a similar facility with or-hints (see documentation topic "hints" in the ACL2 Manual [1]),

except that while ACL2 currently implements or-hints with single-threaded execution, DLP attempts to apply the specified strategies in parallel. Our approach is different from the DLP approach in that, once parallel execution is enabled, we automatically parallelize the proofs of subgoals, without requiring users to mark specific parts of proof attempts for parallel execution. Our approach is also different because we employ *and-parallelism*. As just described, we could extend ACL2(p) to parallelize the use of or-hints, which would be a way to incorporate the ideas of *or-parallelism*.

In 1990, Schumann and Letz presented Partheo, "a sound and complete or-parallel theorem prover for first order logic" [58]. The parallel portion of Partheo's implementation was written in parallel C and was implemented on a network of 16 transputers. This parallel implementation used message passing to run sequential theorem provers [29] based on Warren's abstract machine [3, 61]. Schumann and Letz discuss or-parallelism in Section 4.1 of their paper [58], which is also used in their later work.

The multi-process theorem prover SiCoTHEO is a program that executes multiple SETHEO-based provers in parallel [56]. SiCoTHEO starts multiple copies of the sequential theorem prover SETHEO [29], except with different configuration parameters. Once one of these copies finds the solution to the problem, SiCoTHEO aborts the other copies' searches and returns the result. As already mentioned, ACL2 also provides a way to try different proof strategies, called or-hints. However, ACL2's use of or-hints is not parallelized and distributed over processes.

Wolf and Fuchs discuss two types of theorem proving parallelism: cooperative and non-cooperative [63]. A cooperative approach can use information from one subgoal's proof in the proof of the next subgoal. A non-cooperative approach proves subgoals independently of one another. ACL2's current approach is largely non-cooperative, which makes it a potential target for parallel execution.

Other examples of parallelized theorem provers include Moten's parallel interactive theorem prover MP refiner [39], Maude's concurrent rewriting logic [34], the Peers distributed theorem proving prototype [6], and parallel Isabelle/HOL [33, 62]. The Isabelle/HOL work is best explained with a quotation from one of their papers [33]:

> Isabelle proof documents follow a certain structure that allows various parallel scheduling strategies.... The main observations are as follows.
>
> 1. Large Isabelle applications consist of a DAG-structured collection of theories. Independent nodes in that graph can be loaded in parallel. This is analogous to a parallel make tool....
> 2. Theorem statements are explicit and proofs are irrelevant, in the sense that a theorem can be accepted as correct and used elsewhere without having checked its proof yet. It is, of course, necessary to finish proofs at some point but this can be done independently via futures....

3. Isar proofs have a rich sub-structure, where most runtime is spent in terminal justifications (small local proofs, involving potentially complex automated reasoning tools). Here is a stylized Isar proof text for illustration:

```
lemma A and B
proof
  show A by auto
  show B by blast
qed
```

These `by` steps can be parallelized implicitly, without having to re-implement proof tools like auto or blast involved here.

Our work is different from the above described Isabelle/HOL work in the following ways. The first item above is similar to certifying the ACL2 test suite with the process-level parallelism available via GNU Make's "make -j" feature. As previously discussed, our concern is the interactive time, not the time it takes to certify the test suite. The second item above would be similar to parallelizing the proofs of multiple ACL2 theorems (which was done in 1989, see Section 2.4.2). So, if an ACL2 book defines twenty theorems, users could prove each of those twenty theorems in parallel with one another. Again, this would reduce the time it takes to certify a book, but it would have no bearing on the time it takes to attempt the proof of just one conjecture. The third and final item above would be similar to parallelizing the steps of ACL2's proof checker (see "proof-checker" in the ACL2 Manual [1]), as opposed to the general theorem prover.

We are unaware of any use of parallelism in the proof processes of Coq [5], HOL4 [60], and PVS [41, 59].

2.4.1 Parallelism in the Rewriter of a Basic Boyer-Moore-Style Theorem Prover

One related and interesting application of Multilisp, Qlisp, and Parcel [17] was performed by Harrison and Ammarguellat [18]. In this work, they compared the ability of the Parcel compiler to automatically discover opportunities for parallel execution against the manual use of `future` and `qlet` inside a Boyer-Moore-style rewriter. While they show Parcel to be effective in discovering rewriter-level parallelism opportunities, and they make general comments suggesting a significant speedup, they do not present any statistics. This could be because the emphasis in that work is more about exploring the capabilities of Parcel than it is about exploring the efficiency of parallelizing a Boyer-Moore-style prover at the level of the rewriter.

2.4.2 Parallelism in ACL2

Process-level parallelism is currently available to ACL2 users via GNU Make's "make -j" option. This option allows the certification of ACL2 libraries across multiple cores on one machine. While this option optimizes the most common benchmark for improving ACL2 performance, it does nothing to improve the interactive delay involved in using the ACL2 theorem prover. It is this interactive delay that this work addresses and improves.

In 1989, theorem-level parallelism was made available to NQTHM (the

predecessor of ACL2) by Kaufmann and Wilding [26]. The focus of their work was on quickly proving all of the theorems in a file in parallel, as a batch check when one already believed that the proofs would succeed. Using this approach they obtained about two-thirds of the theoretical speedup. However, this approach does not reduce the time it takes to develop proofs in an interactive environment, which is the focus of our work.

Additionally, we have previously fully implemented and integrated four parallelism primitives designed to allow ACL2 users to evaluate expressions in parallel: `plet`, `pargs`, `pand`, and `por` [43, 44, 45]. In an effort to support proof parallelism, this dissertation further extends ACL2 to support a parallelism primitive named `spec-mv-let`.

Chapter 3

Our Application: ACL2

Our target application is ACL2. In this chapter we explain ACL2's main proof process, called the *waterfall*, and how we parallelize the execution of the waterfall. We also introduce the mechanism that ACL2 provides for modeling changes to the global program state in a functional manner. Finally, we introduce the subset of ACL2 output most relevant to debugging failed proof attempts, called *proof checkpoints*.

3.1 ACL2's Main Proof Process: The *Waterfall*

The main ACL2 proof process is called the *waterfall* (see Figure 3.1 for a visual representation of this process, adapted from *An ACL2 Tutorial* [25]). We have parallelized the execution of the methods that perform simplification, destructor elimination, fertilization, generalization, and the elimination of irrelevance. Since induction occurs outside the waterfall, we do not parallelize the use of induction. However, once an induction scheme is chosen and applied, ACL2 will again enter the waterfall, and parallel execution can resume. Parallelizing the proof process at the waterfall level results in *subgoal-level parallelism*. By parallelizing the proof process at this level (as opposed

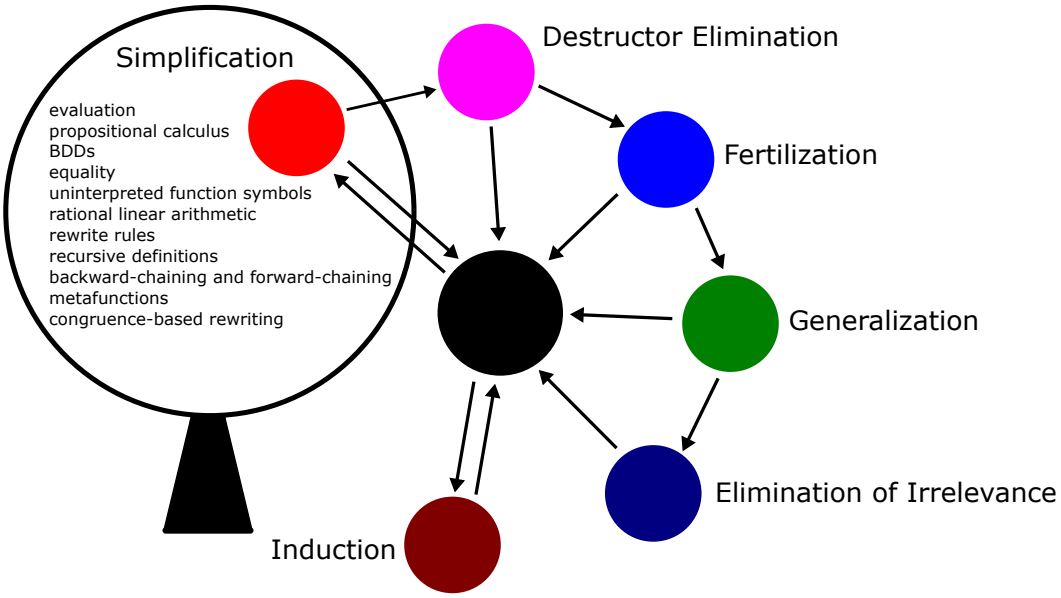

Figure 3.1: The ACL2 Waterfall

to parallelizing the ACL2 rewriter or another function within the waterfall), the overhead associated with parallel execution is rendered insignificant (see Section 8.3.1 for further discussion of this claim).

3.1.1 Waterfall Implementation

Another view of ACL2's main proof process comes from examining the functions themselves and their associated call-graph, shown in Figure 3.2 (see the end of this section for an explanation of the call-graph). When ACL2 begins a proof, it starts by calling the function `prove-loop2`, which takes many arguments, including a list of clauses to prove (a clause is a set of implicitly disjoined terms). `Prove-loop2`'s specification is simple: call the

```
prove-loop2
  waterfall
     waterfall1-lst
        waterfall1
           waterfall0
              waterfall-step
              waterfall0 (optional)
              waterfall1-lst (optional)
        waterfall1-lst
  prove-loop2
```

Figure 3.2: Function Call Graph for the ACL2 Waterfall

function `waterfall` repeatedly until the necessary set of clauses is proven, or `prove-loop2`'s heuristics indicate that it is time to admit failure. `Prove-loop2` keeps track of whether the current call of the waterfall is the first call of the waterfall (typically called the "top-level"), a call after deciding to perform induction, a call after deciding to perform sub-inductions, or part of performing proofs associated with a "forcing round" (see documentation topic "forcing-round" in the ACL2 manual [1] for an explanation of forcing rounds). Each recursive call of `prove-loop2` moves the proof process from one of these rounds to the next (e.g., from the "top-level" round to the "induction round").

As shown in the figure, `prove-loop2` calls function `waterfall`, passing in the same list of clauses that it was given. When at the top-level, there is exactly one clause in this clause list, but this list can contain more clauses in subsequent rounds. `Waterfall` is non-recursive and just serves as a wrapper to `waterfall1-lst`.

`Waterfall1-lst` also accepts a list of clauses to prove (in the ACL2 vernacular, each clause given to the waterfall is called a *subgoal*). In the serial version of ACL2, `waterfall1-lst` calls `waterfall1` on the first element in that list of clauses, and then it calls itself recursively on the remainder of the list. If `waterfall1-lst` is given an empty list of clauses, it just returns. If `waterfall1-lst` and its subfunctions (including `waterfall0` and `waterfall-step`, which implement the heuristics shown in Figure 3.1) are able to prove each input clause, then ACL2 will consider the clauses to have been proven and display "Q.E.D." If `waterfall1-lst` is unable to prove each of the input clauses (perhaps because it decides to postpone some proof obligations), then `prove-loop2` will attempt to prove the remaining clauses (with subsequent calls of `waterfall`).

The call graph for the waterfall can be found in Figure 3.2. The indentations represent calls to the indented function name from the function name relative to its indent. As an example, `waterfall1-lst` is called from `waterfall`, and since no other function names appear at the same indentation level as that call of `waterfall1-lst`, the reader can derive that `waterfall` calls no other function that we deem relevant to understanding how we parallelize the waterfall. `Waterfall1-lst` is shown as calling itself, because it recurs on itself.

3.1.2 The Waterfall's Potential for Parallel Execution

After examining the waterfall, we can see one place in particular where it could be good to introduce parallel execution: the function `waterfall1-lst`. In the serial version, `waterfall1-lst` calls `waterfall1` on the first element of the clause list given to it, and then it recurs on the rest of the clause list. Executing the call to `waterfall1` and the recursive call of `waterfall1-lst` in parallel has the potential to provide the benefits outlined in Section 1.2. We achieve such parallel execution by inserting a call of our parallelism primitive `spec-mv-let` (which is explained in Section 4.2.2.5) into the definition of `waterfall1-lst`. Additionally, since we are parallelizing the proofs of subgoals, there is potential to present failed subgoal proofs to users much more quickly than if we were executing serially. We further discuss these two benefits in Chapter 7.

One other difference between the serial version of the waterfall and the parallel version is the way that we split up the list of clauses given to `waterfall1-lst`. In the serial version, the list of clauses is processed as described above, one at a time. However, in the parallel version, `waterfall1-lst` (really, the function is named `waterfall1-lst@par`, but we disregard the naming difference and refer to `waterfall1-lst`) splits the list into halves and recurs on both halves of the list. It is not until `waterfall1-lst` is called on a list of a single element that `waterfall1` is called. This hierarchical approach more efficiently uses the underlying parallelism resources and is further discussed near the end of Section 8.3.2.3.

3.1.3 Introduction to the Parallelism Work Queue and Worker Threads

As mentioned in Section 3.1.2, our strategy for executing the waterfall in parallel includes inserting our parallelism primitive `spec-mv-let` into the parallel version of `waterfall1-lst`. Described more fully in Section 4.2.2.5, `spec-mv-let` has both an "eager" and a "speculative" component to it. In the top-most recursive call of `waterfall1-lst`, the eager component will be the proofs for the first half of the clause list and the speculative component will be the proofs for the second half of the clause list. The eager part (proving the first half of the clause list) will be processed by the current thread, and the speculative computation (proving the second half of the clause list) will be bundled up and placed on a global parallelism *work queue* for execution by another thread. We refer to this work queue throughout the dissertation, and we refer to the threads that process this queue as *worker threads*. Further details can be found in Chapter 8.

3.2 Introduction to ACL2's Model for Functions with Side-effects

The programming language of ACL2 has purely functional semantics. However, ACL2 provides a mechanism that permits one to program in a style that "feels" imperative in nature. For example, ACL2 includes tricks that make it efficient to manipulate the global program state. When run in parallel, such global modifications can create race conditions that cause problems for

the user, the theorem prover, or both. Providing a mechanism that tracks such global modifications makes removing them from parallelized portions of the theorem prover much easier. The mechanism that models the execution of code with side-effects in a functional manner is an ACL2 variable named *state*. Throughout the dissertation, *state* is italicized when we are referring to this variable (as opposed to the global program state, which is related but still technically distinct). *State* is used to track changes to the global program state in the following way.

Whenever users modify the global state of the program (perhaps by writing to a global variable or performing some I/O operations), part of the return value for the function that performs the modification must include *state*. Thus, if a function nested very deeply in the call stack modifies the global state of the program, we will know at the highest level of the call stack that such a modification has occurred. This is because *state* will have been returned by each function within the call stack.

Due to this return value signature, it is relatively easy to identify areas of the ACL2 code and user-level code that are inherently single-threaded in nature. One of the tasks of our project involved modifying the subfunctions of the waterfall that returned *state* so that they no longer needed to modify the global program state, thus removing the need to return *state*. We also disallow the use of functions that continue to return *state*, causing a clear error whenever these functions are encountered under parallel execution.

3.3 Introduction to ACL2's Proof Output

The amount of output that ACL2 can provide is typically overwhelming when printed in an interleaved manner. Even if the theorem prover were to atomically print one proof subgoal at a time, with the non-deterministic order of printing, users would be unable to find the right subgoal upon which to focus. As such, it is potentially useful to find a subset of the output that is most relevant to users' proof discovery and to print only that portion.

Proof Attempt Checkpoints Work has already been done in the serial version of ACL2 to create a mode called *gag mode* (see documentation topic "gag-mode" inside the ACL2 Manual [1]) that only prints subgoals that have not been proven with most of the available proof heuristics (simplification, destructor elimination, fertilization, generalization, and elimination of irrelevance). These unproven subgoals that gag mode prints are called *key checkpoints*. ACL2 users typically direct their focus at these very subgoals, and ACL2(p) reuses the ideas behind gag mode to provide output similar to these key checkpoints. The way that ACL2(p) provides this restricted set of output is further discussed in Section 6.4.

3.4 A Warmup Example

In this section we present an example to warmup the reader, theorem *ideal-4-way*. The goal of this example is to familiarize the reader with the structure of ACL2 proofs and to introduce the timing information that is

```
(defthm ideal-4-way
  (and (f1 x) (f2 x) (f3 x) (f4 x))
  :otf-flg t)
```

Figure 3.3: Theorem *Ideal-4-way*

available for printing when performing a proof in parallel. Theorem *ideal-4-way* is shown in Figure 3.3. The proof simply involves calling functions (named *f1* through *f4*), that count down from a very large number and test that the value returned is not equal to a particular constant. Each of the subgoals provided in the proof is proved by execution, and the overhead for performing this proof in parallel is insignificant.

Figure 3.4 shows the flow for this proof. From this picture, we can determine that *Goal* (the top-level subgoal) is broken down into four subgoals: *Subgoal 4*, *Subgoal 3*, *Subgoal 2*, and *Subgoal 1*. Note that ACL2 proves subgoals in reverse numeric order. *Goal* takes 522 microseconds before it splits into these four subgoals, at which point *Subgoal 4* begins processing. *Subgoal 4* takes about 37 seconds to finish, and then *Subgoal 3*'s proof starts. *Subgoal 3* also takes about 37 seconds, and then *Subgoal 2* and *Subgoal 1* also eventually start and complete in about 37 seconds each.

When we run the proof with `pseudo-parallel` waterfall parallelism (a single-threaded mode that is explained in Section 6.1), a setting of `limited` for the waterfall parallelism output (explained in Section 6.4.1) and a non-`nil` value stored in the ACL2 global variable `waterfall-printing-when-finished`, we receive the output shown in Figure 3.5 (note that throughout the dissertation, we will refer to Lisp "keywords", such as the aforementioned

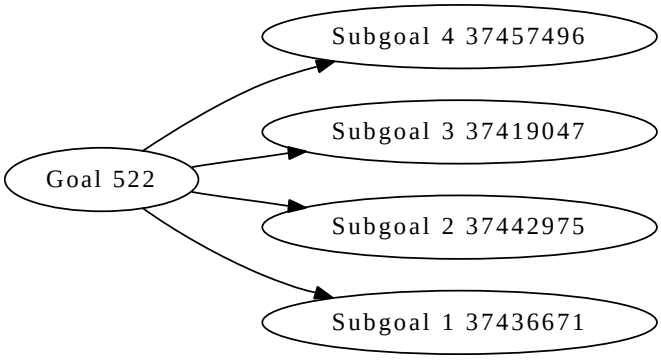

Figure 3.4: Proof Dependency Tree for Theorem *Ideal-4-way* (times shown in microseconds)

```
At time 0.000159 sec, starting: Goal
At time 0.000616 sec, starting: Subgoal 4
At time 37.433770 sec, finished: Subgoal 4
At time 37.433926 sec, starting: Subgoal 3
At time 74.845300 sec, finished: Subgoal 3
At time 74.845460 sec, starting: Subgoal 2
At time 112.256900 sec, finished: Subgoal 2
At time 112.257040 sec, starting: Subgoal 1
At time 149.668370 sec, finished: Subgoal 1
At time 149.668500 sec, finished: Goal
```

Figure 3.5: Timing Information for Serially Proving Theorem *Ideal-4-way*

`pseudo-parallel`, and although the technical name of the keyword contains a colon, as in `:pseudo-parallel`, we typically omit the colon to improve readability). The reported times are slightly different than those shown in Figure 3.4, because they are taken from different runs of the proof. Note that *Goal* is not marked as finished until after each of the subgoals finish because *Goal* depends on them.

When we run the same proof with almost the same settings, but with `full` waterfall parallelism (a parallelism mode that is explained in Section 6.1),

```
At time 0.004642 sec, starting: Goal
At time 0.005149 sec, starting: Subgoal 4
At time 0.005884 sec, starting: Subgoal 1
At time 0.006233 sec, starting: Subgoal 2
At time 0.006633 sec, starting: Subgoal 3
At time 37.406784 sec, finished: Subgoal 2
At time 37.432980 sec, finished: Subgoal 4
At time 37.435146 sec, finished: Subgoal 1
At time 37.449608 sec, finished: Subgoal 3
At time 37.449806 sec, finished: Goal
```

Figure 3.6: Timing Information for Proving Theorem *Ideal-4-way* in Parallel

we receive the output shown in Figure 3.6. Notice that the proof now finishes in about 37 seconds instead of 150 seconds, and that the reported timing information indicates that all four of the subgoals were executing concurrently. This is how the timing output for parallelized proofs looks (there are no checkpoints printed because there are no subgoals that fail and need examination from the ACL2 user). After examining figures 3.4, 3.5, and 3.6, the reader should be comfortable with reading subgoal dependency trees and the timing output that is available for printing when using waterfall parallelism.

Chapter 4

Parallelism Primitives

This chapter introduces the ACL2(p) programming primitives that enable ACL2 programs to execute in parallel. We categorize these features into two groups: (1) those similar to the primitives commonly found in a multi-threading library like POSIX Threads [20] and (2) those that offer a level of abstraction that does not involve reasoning about locks, signaling mechanisms, and threads. Category (2) is further broken down into: (A) abstractions that are only available by escaping to the Lisp mode of ACL2 and (B) abstractions that are also embedded in and available from the ACL2 logic and programming language. For the sake of discussion in this chapter, we use the term "low-level Lisp multi-threading primitives" to refer to (1), we use "high-level Lisp parallelism primitives" to refer to (2A), and we use "ACL2 parallelism primitives" to refer to (2B).

It is helpful to think of the layers of parallelism primitives as a stack. Figure 4.1 shows the relationship between the different layers of functionality. The set of primitives colored in light-red are only accessible by escaping to the host Lisp, and the set of primitives colored in light-green are embedded in and accessible from the ACL2 logic and programming language. As shown

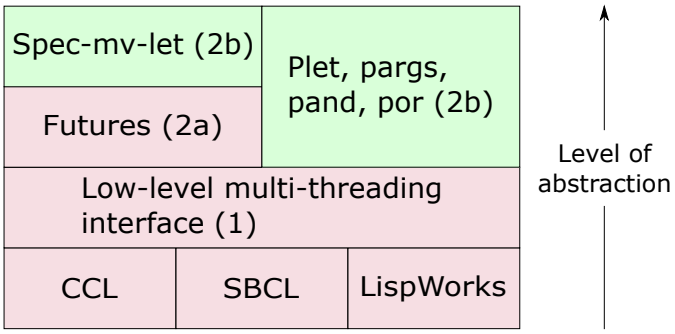

Figure 4.1: Parallelism Primitive Stack

in the figure, we build our low-level multi-threading interface upon the primitives provided by CCL, SBCL, and LispWorks. This multi-threading interface unifies the primitives that each Lisp implementation provides into one set of primitives, which makes our higher level primitives more easily maintained. This multi-threading interface is detailed in Section 4.1.

Another layer of abstraction provided only at the Lisp-level is our futures library. This futures library provides three primitives for creating, reading, and aborting futures and is detailed further in Section 4.2.1. We then use the futures library to implement the logic-level ACL2 parallelism primitive that we use to parallelize the waterfall, `spec-mv-let` (described in Section 4.2.2.5). Building `spec-mv-let` upon the futures library makes it more easily maintained than if it were built directly upon the low-level multi-threading interface.

Our final set of parallelism primitives, `plet`, `pargs`, `pand`, and `por`, are also available from the ACL2 logic (and described in Section 4.2.2). These

primitives were built before we had the futures library, and as such, are built directly upon the low-level multi-threading interface. In fact, the implementations of `pand` and `por` support early termination, and they could not be implemented on top of our current implementation of futures. In an effort to make them more maintainable and possibly more efficient, `plet` and `pargs` could be rewritten to use the futures library. However, we leave that potential task as future work.

4.1 Low-level Lisp Multi-threading Primitives

ACL2(p) contains an interface that unifies different low-level Lisp multi-threading primitives into one set of functions and macros. This interface provides mechanisms for mutual exclusion, signaling, and controlling threads. More specifically, it implements semaphores, condition variables, locks, and the ability to start, interrupt, monitor, and kill threads.

Since Clozure Common Lisp (CCL) [9], Steel Bank Common Lisp (SBCL) [55], and LispWorks [30] provide primitives sufficient to implement these mechanisms in a relatively straightforward manner, our interface supports these three Lisps. In addition to making ACL2(p) available for users that use a Lisp different than our Lisp of choice, using multiple Lisp implementations helped us ascertain whether bugs were Lisp implementation bugs or caused by problems in ACL2(p)'s code. Another benefit of using multiple Lisp implementations is that our code has been checked by more than one compiler, allowing us to catch bugs that any one of the Lisp implementations

might mask.

4.1.1 Mutual Exclusion and Signaling

To satisfy the need for mutual exclusion and signaling mechanisms, our interface provides locks, semaphores, and condition variables. Since all three Lisps support locks, implementing the locking interface is only a matter of calling the CCL, SBCL, and LispWorks equivalents. Implementing semaphores is a little more complicated. While CCL has provided a feature for semaphores called a *semaphore notification object* (described in the next paragraph) for many years, SBCL has only recently implemented this feature, and LispWorks does not yet provide the feature. As such, we must implement semaphore notification objects for SBCL and LispWorks; instead of just being a wrapper for the corresponding Lisp call, our semaphore implementation in SBCL and LispWorks uses a data structure that contains a counter, a lock, and a condition variable. For further implementation details, the reader can reference file *multi-threading-raw.lisp*, which is distributed with ACL2 [1]. Our interface also provides condition variables by calling the SBCL and LispWorks equivalents, and our condition variable implementation on CCL is adapted from the Bordeaux Threads project [7].

Guaranteeing safety and liveness properties in our parallelism primitives requires a way to test whether a semaphore signal was received. Under normal thread execution, this only involves checking the return value of the `wait-on-semaphore` function. However, if the execution of a thread is aborted

while it is waiting on a semaphore, there is no return value to check. In support of this project, the CCL maintainers created a *semaphore notification object* that, when provided, is set atomically with the receipt of a semaphore's signal. Often when a thread receives a signal from a semaphore, it needs to react by doing something (e.g., recording that the thread is about to transition from being idle to actually using a CPU core and executing a piece of parallelism work). By placing the appropriate action and the clearing of the notification object inside a `without-interrupts`, the program is guaranteed that the semaphore notification object is cleared if and only if that action was performed. In the event that a particular execution is aborted, the surrounding `unwind-protect` (see next paragraph) can check the notification object and determine whether that action was performed. And, if the action was not performed, the cleanup portion of the `unwind-protect` can perform the action at that point. In this way, the parallelism library uses semaphore notification objects to keep an accurate record of threads, pieces of parallelism work, and other data.

A Lisp `unwind-protect` takes two sets of arguments: a body and cleanup forms. After the body completes, the cleanup forms execute. Even in the event that an error occurs, the cleanup forms will always run. Furthermore, in our version of `unwind-protect` (named `unwind-protect-disable--interrupts-during-cleanup`), interrupts are disabled while executing the cleanup forms. These properties are important to successfully implementing our higher-level parallelism primitives.

While using semaphores and condition variables is almost always more efficient than a busy wait (see Section 5.2.1 in *New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools* [15]), our multi-threading interface also provides the function `thread-wait`, which allows a thread to busy wait. A thread calls this function by passing in a function and argument to execute. If this execution returns a non-`nil` result, `thread-wait` returns and the thread unblocks. Otherwise, the thread enters a loop, where it sleeps for a small amount of time (for example, 50 milliseconds in SBCL and LispWorks) and then re-applies the given function to the appropriate argument or arguments. The thread exits this loop once the application of the function returns a non-`nil` result.

4.1.2 Controlling Threads

Our multi-threading interface allows a programmer to create, interrupt, monitor, and kill threads. Since CCL, SBCL, and LispWorks all support thread creation, interruption, and termination, our ACL2 implementations of these functions only serve as wrappers for the underlying Lisp primitives.

4.1.3 Low-level Lisp Multi-threading Primitive Dictionary

Included in the above described multi-threading library are the following functions and macros, which have the provided input and output signatures and described properties.

Without-interrupts

Inputs: *forms*

Output: *result-of-executing-forms*

Description: Prevents execution of any of the surrounded forms from being interrupted. This behavior takes priority over any outer call of `with-interrupts`. This being said, since we do not have a good use case for providing a version of `with-interrupts`, we omit it from our interface. Returns the result of executing the given forms.

Unwind-protect-disable-interrupts-during-cleanup

Inputs: *body, cleanup-forms*

Output: *result-of-executing-body*

Description: Is the same as the Common Lisp [42] `unwind-protect` but provides an additional guarantee that the cleanup-forms cannot be interrupted and are always executed. Returns the result of executing the given body.

Make-atomically-modifiable-counter

Inputs: [none]

Output: *counter*

Description: Returns a counter that can be atomically incremented and decremented. This constructor is necessary because not all of the Lisps support atomically incrementing or decrementing integer variables without placing the variables inside a structure.

Atomically-modifiable-counter-read

Inputs: *counter*

Output: *integer*

Description: Reads the value from an atomic counter.

Atomic-incf

Inputs: *counter*

Output: *integer*

Description: Atomically increments a counter. Returns the new value of the counter. We could implement this atomicity in a variety of ways, perhaps by using locks, `without-interrupts`, compare and swap instructions, etc. Regardless of the implementation, by "atomic", we mean that if two threads simultaneously call `atomic-incf`, that the counter will be reliably incremented by two. Also, if two threads simultaneously call `atomic-incf` and `atomic-decf`, the end value will be the same as the original value.

Atomic-decf

Inputs: *counter*

Output: *integer*

Description: Atomically decrements a counter. Returns the new value of the counter. We could implement this atomicity in a variety of ways,

perhaps by using locks, `without-interrupts`, compare and swap instructions, etc. Regardless of the implementation, by "atomic", we mean that if two threads simultaneously call `atomic-decf`, that the counter will be reliably decremented by two. Also, if two threads simultaneously call `atomic-incf` and `atomic-decf`, the end value will be the same as the original value.

Make-lock

Inputs: [none]

Output: *lock*

Description: Returns a recursive lock that can be used to provide the guarantee of mutually exclusive execution. A recursive lock provides the property that the same thread can obtain the same lock more than once (without releasing it in the interim), and the thread will not deadlock (see Chapter 8 in *Programming with UNIX Threads* [40] for further information on recursive locks).

Deflock

Inputs: *lock-symbol*

Output: *macro-symbol*

Description: Defines both a Lisp lock object and a macro for using a lock with the name given by the provided symbol. While all of the other primitives described in this section are defined for use only within the

Lisp mode of ACL2, the macro that `deflock` defines can be used within the ACL2 logic. The first argument to `deflock` is a symbol that serves as the <*lock-name*> in the macro, which will be named `with-`<*lock-name*>. If a user attempts to define the same lock symbol twice or more, the second and all subsequent definitions will be redundant. Returns the symbol for the name of the newly defined macro.

With-lock

Inputs: *lock, forms*

Output: *result-of-executing-forms*

Description: Grabs the given lock, blocking until it is acquired; executes the given forms; and then releases the lock. This primitive provides mutually exclusive execution. Returns the result of executing the given forms.

With-<*lock-name*>

Inputs: *forms*

Output: *result-of-executing-forms*

Description: Grabs the lock named <*lock-name*> (as defined by `deflock`), blocking until it is acquired; executes the given forms; and then releases the lock. While all of the other primitives described in this section are defined for use only within the Lisp mode of ACL2, this macro, defined by `deflock`, can be used within the ACL2 logic to provide mutually exclusive execution (for example, there is a macro named

`with-output-lock` that is useful when executing ACL2 code that performs output). Returns the result of executing the given forms.

Run-thread

Inputs: *thread-name, function, arguments*

Output: *thread-object*

Description: Applies the given function to the given arguments. This application occurs in a fresh thread with the given name.

Interrupt-thread

Inputs: *thread-object, function, arguments*

Output: `nil`

Description: Interrupts the given thread and then, in that thread, applies the given function to the given arguments. When this function application returns, the thread resumes from the interrupt (from where it left off).

Kill-thread

Inputs: *thread-object*

Output: `nil`

Description: Kills the given thread.

All-threads

Inputs: [none]

Output: *list-of-thread-objects*

Description: Returns a list of all Lisp thread objects that currently exist in the system.

Current-thread

Inputs: [none]

Output: *thread-object*

Description: Returns the thread object associated with the calling thread.

Thread-wait

Inputs: *function, arguments*

Output: `nil`

Description: Provides an (inefficient) mechanism for the current thread to wait until a given condition, defined by the application of the given function to the given argument or arguments, is true. When performance matters, we advise using a signaling mechanism instead of this relatively high latency function.

Make-condition-variable

Inputs: [none]

Output: *condition-variable*

Description: Returns a condition variable that can be used to send signals between threads. A thread can "wait on" a condition variable, and

it will block until that condition variable is "signaled" by another thread. Unlike semaphores, condition variables store no state. A condition variable is simply a mechanism for maintaining a queue of threads that are blocked, waiting for the condition variable to be signaled, and should be unblocked when the condition variable is signaled. Using a condition variable is typically much faster than using a busy wait (i.e., waiting in a loop for a condition to be true or calling `thread-wait`).

Signal-condition-variable

Inputs: *condition-variable*

Output: `nil`

Description: Signals the given condition variable once, causing any one thread that is waiting for the given signal to resume execution.

Broadcast-condition-variable

Inputs: *condition-variable*

Output: `nil`

Description: Signals the given condition variable in a manner such that all threads waiting for a signal from that condition variable will resume execution. Not supported in CCL.

Wait-on-condition-variable

Inputs: *condition-variable, lock*

Output: `t`

Description: Suspends execution of the calling thread until another thread signals the given condition variable by calling `signal-condition--variable` or `broadcast-condition-variable`. Returns `t` (true) once the signal is received.

Make-semaphore

Inputs: [none]

Output: *semaphore*

Description: Returns a counting semaphore, which is a data structure that includes a field containing a natural number, named the "count" field. A thread can "wait on" a counting semaphore, and it will block in the case that the semaphore's count is 0 (a semaphore's count can not go below 0). To "signal" such a semaphore means to increment that field and to notify a single thread "waiting" on that semaphore that the semaphore's count has been incremented. Then this thread, which is said to "receive" the signal, decrements the semaphore's count and is unblocked. Using a semaphore is typically much faster than using a busy wait (i.e., waiting in a loop for a condition to be true or calling `thread-wait`).

Make-semaphore-notification

Inputs: [none]

Output: *semaphore-notification-object*

Description: Returns a semaphore notification object used to record whether a semaphore signal has been received (for use with `wait-on--semaphore`). Semaphore notification objects are necessary, because a thread can be interrupted at any point in time, and the thread must be able to determine whether it is no longer waiting on a semaphore because it received the signal or because computation was aborted.

Semaphore-notification-status

Inputs: *semaphore-notification-object*

Output: *boolean*

Description: Reads a semaphore notification object, returns `t` (true) if that semaphore was signaled, and returns `nil` if it was not.

Signal-semaphore

Inputs: *semaphore*

Output: `nil`

Description: Increments a semaphore's count by one and, if such a thread exists, causes exactly one thread waiting on that semaphore to resume execution (the resuming thread will then decrement the semaphore's count). If there is no thread waiting on that semaphore, then no thread resumes execution due to the call to `signal-semaphore`.

Wait-on-semaphore

Inputs: *semaphore, timeout, semaphore-notification-object*

Output: *boolean*

Description: If the given semaphore's count is larger than zero, `wait--on-semaphore` decrements the count, records receipt of the signal in an optionally provided semaphore notification object, and returns `t` (true). If the given semaphore's count is zero, `wait-on-semaphore` suspends the execution of the current thread and blocks until the semaphore is signaled. Alternatively, if there is a timeout given, the current thread will only block until the amount of time specified in the timeout argument elapses. If the call to `wait-on-semaphore` receives a signal, it decrements the semaphore's count, records receipt of the signal in an optionally provided semaphore notification object, and returns `t` (true). In the event that the call to `wait-on-semaphore` times out, the thread unblocks and `wait-on-semaphore` returns `nil`.

Initial-threads

Inputs: [none]

Output: *list-of-thread-objects*

Description: Returns a list of thread objects that are part of the underlying CCL, SBCL, or LispWorks runtime system and not part of our parallelism system.

All-threads-except-initial-threads-are-dead

Inputs: [none]

Output: *boolean*

Description: Returns `t` (true) if all of the threads created by our parallelism system have terminated. Otherwise, it returns `nil`. This function is used to help determine when the parallelism system has been reset to a stable state.

4.2 High-level Lisp and ACL2 Parallelism Primitives

This section describes eight parallelism abstractions available in ACL2(p). The first three constitute our futures library and are not available from within the ACL2 logic and are only available by escaping to the host Lisp. The other five abstractions, `plet`, `pargs`, `pand`, `por`, and `spec-mv-let`, are available for use both in Lisp and the ACL2 logic.

4.2.1 Futures Library

There are three high-level Lisp-only parallelism primitives that enable and control parallel execution: `future`, `future-read`, and `future-abort`. The `future` macro surrounds a form and returns a data structure with fields including the following: a closure representing the given computation, a slot (initially empty) for the value returned by that computation, and a mechanism for knowing when that slot's value is valid. This structure can then be passed

to `future-read`, which will access the `value` field after the closure finishes executing. The final primitive, `future-abort`, terminates futures whose values are no longer needed.

The naïve version of the Fibonacci function, shown in Figure 4.2, illustrates the use of `future` and `future-read`.

```
(defun pfib (x)
  (if (< x 33)
      (fib x)
    (let ((a (future (pfib (- x 1))))
          (b (future (pfib (- x 2)))))
      (+ (future-read a)
         (future-read b)))))
```

Figure 4.2: Definition of Fibonacci Using Futures

The implementation of `future` provides the following behavior: when a thread executes a call of `future`, it returns a future, *F*. *F* contains a closure that is placed on the *work queue* for evaluation by a *worker thread* (further explained in Section 8.2). The value returned by that computation may only be obtained by calling the `future-read` function on *F*. If a thread tries to read *F* before the worker thread finishes executing the closure, the reading thread blocks until the worker thread finishes. The final primitive, `future-abort`, removes a given future, *F*, from the work queue, sets a flag in *F* to record the abortion, and aborts execution (if in progress) of *F*'s closure.

4.2.2 Parallelism Primitives Available from the ACL2 Logic

This section describes the five parallelism primitives added to the ACL2 logic as part of ACL2(p). The first four primitives, `plet`, `pargs`, `pand`, and `por`

are also covered in the ACL2 Manual [1], Rager's Master's Thesis [44], and in Rager and Hunt's *Implementing a Parallelism Library for a Functional Subset of Lisp* [45]. The fifth and newest primitive, `spec-mv-let` is documented in the ACL2 Manual [1] and Rager, Hunt, and Kaufmann's *A Futures Library and Parallelism Abstractions for a Functional Subset of Lisp* [46].

The design of our ACL2 parallelism primitives is driven by two goals. First, users need a way to parallelize computation efficiently. Second, the use of our ACL2 parallelism primitives should be as transparent to the theorem prover as possible. Thus, each ACL2 function has two definitions: (1) the version for the ACL2 logic, used to prove theorems and (2) the Lisp version, used for efficient execution. The logical version avoids complicating reasoning with the complexity of low-level multi-threading primitives. If one assumes that the Lisp version of the function is implemented correctly, then the Lisp and logical definitions are functionally equivalent.

The first four logic-level abstractions, `plet`, `pargs`, `pand`, and `por`, are built directly upon our low-level multi-threading primitives. When we created `spec-mv-let`, we decided to build upon an intermediate level of abstraction (i.e., futures). As a result, the `spec-mv-let` implementation is more easily maintained than the earlier four primitives.

4.2.2.1 Plet

The first ACL2 parallelism primitive, `plet`, is logically equivalent to the macro `let`. However, in its Lisp version, `plet` can execute the computations

for its bindings in parallel and then apply a closure created from the body of the `plet` to the results of those executions. A simple example use of `plet` is shown in Figure 4.3.

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((zp x) 0) ; test whether x <= 0
        ((= x 1) 1)
        (t (plet
            (declare (granularity (> x 30)))
            ((fib-x-1 (pfib (- x 1)))
             (fib-x-2 (pfib (- x 2))))
            (+ fib-x-1 fib-x-2)))))
```

Figure 4.3: Definition of Fibonacci Using `Plet`

In this example, the executions for the values of `fib-x-1` and `fib-x-2` occur in parallel, and then the closure containing the call of the macro `+` is applied to `fib-x-1` and `fib-x-2`. We use a closure so that macros can be used in the body of a `plet`.

In order to minimize the occurrence of parallelism overhead, it is desirable to use `plet` in calls whose bindings require a large amount of time to compute. In support of this idea, a *granularity form* (see Section 4.3 in Rager and Hunt [45] for further details on our granularity forms) may be used to avoid parallelizing computations that take shorter amounts of time.

4.2.2.2 Pargs

The second ACL2 parallelism primitive, `pargs`, is logically the identity macro. `Pargs` surrounds a function call whose arguments it may evaluate in parallel and then applies the function to the results of those parallel argument

evaluations. A simple example use is shown in Figure 4.4. In this example, the Lisp version can execute arguments `(pfib (- x 1))` and `(pfib (- x 2))` in parallel, and then the function `binary-+` will be applied to the list formed from the results of those two executions.

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((<= x 0) 0)
        ((= x 1) 1)
        (t (pargs
            (declare (granularity (> x 30)))
            (binary-+ (pfib (- x 1))
                      (pfib (- x 2)))))))
```

Figure 4.4: Definition of Fibonacci Using `Pargs`

It is an error to apply `pargs` to macro calls (such as the macro `+`) because macros do not evaluate their arguments. However, macro calls can often be handled using the aforementioned `plet`.

4.2.2.3 Pand

The third ACL2 parallelism primitive, `pand`, evaluates its (zero or more) arguments in parallel and returns their conjunction as a Boolean result. This execution differs from the execution of a corresponding call of `and` in two ways. First, `pand` returns a Boolean result. This Booleanization makes it consistent with `por`, which is described in the next section. The second difference is that `pand` is not lazy; that is, the second argument can be executed even if the first argument evaluates to `nil`. Why does this matter? Consider the following call:

```
(pand (consp x)
      (equal (car x) 'foo))
```

With `pand` replaced by `and`, the falsity of `(consp x)` prevents the execution of `(car x)`. This is different from `pand`, where both `(consp x)` and `(equal (car x) 'foo)` can execute in parallel.

As an example, suppose that the function `valid-tree` traverses a tree to test that each atom is a `valid-tip`. A parallel version of `valid-tree` could be defined as shown in Figure 4.5.

```
(defun valid-tree (x)
  (declare (xargs :guard t))
  (if (atom x)
      (valid-tip x)
    (pand (valid-tree (car x))
          (valid-tree (cdr x)))))
```

Figure 4.5: Definition of `Valid-tree` Using `Pand`

Once one of the `pand` arguments evaluates to `nil`, the `pand` call can immediately return `nil`. This optimization for `pand` (and as mentioned below for `por`) is called *early termination* and described in Section 4.3 of Rager and Hunt's *Implementing a Parallelism Library for a Functional Subset of Lisp* [45].

4.2.2.4 Por

The fourth ACL2 parallelism primitive, `Por`, executes its arguments in parallel, computes their disjunction, and returns a Boolean result. Since the execution order of each argument becomes nondeterministic when executed in parallel, returning a Boolean value is important in order to avoid different

```
(spec-mv-let
 (v1 ... vn)  ; bind distinct variables
 <spec>       ; execute speculatively; return n values
 (mv-let
  (w1 ... wk) ; bind distinct variables
  <eager>     ; execute eagerly
  (if <test>
      <typical-case> ; may mention v1 ... vn
    <abort-case>)))  ; may not mention v1 ... vn
```

Figure 4.6: Form for Using `Spec-mv-let`

results for the same `por` call. To avoid always having to execute the arguments necessary to obtain the left-most `non-nil` argument result, the result is simply converted to a Boolean value. Analogous to `pand`, a non-`nil` evaluation result from an earlier argument to `por` does not necessarily prevent execution of later arguments, but early termination can abort execution of irrelevant arguments once a non-`nil` argument value is found.

4.2.2.5 Spec-mv-let

We built the `spec-mv-let` ACL2 parallelism primitive on top of the three futures primitives. Creating `spec-mv-let` avoids the potentially difficult task of introducing futures into the ACL2 programming language and logic. `Spec-mv-let` is similar to `mv-let` (a mechanism for returning more than one value as part of a function's return signature, and also ACL2's notion of Lisp's `multiple-value-bind`). Our design of `spec-mv-let` is guided by the shape of the code where we parallelize ACL2's proof process. `Spec-mv-let` calls have the form shown in Figure 4.6.

Execution of the above form proceeds as suggested by the comments

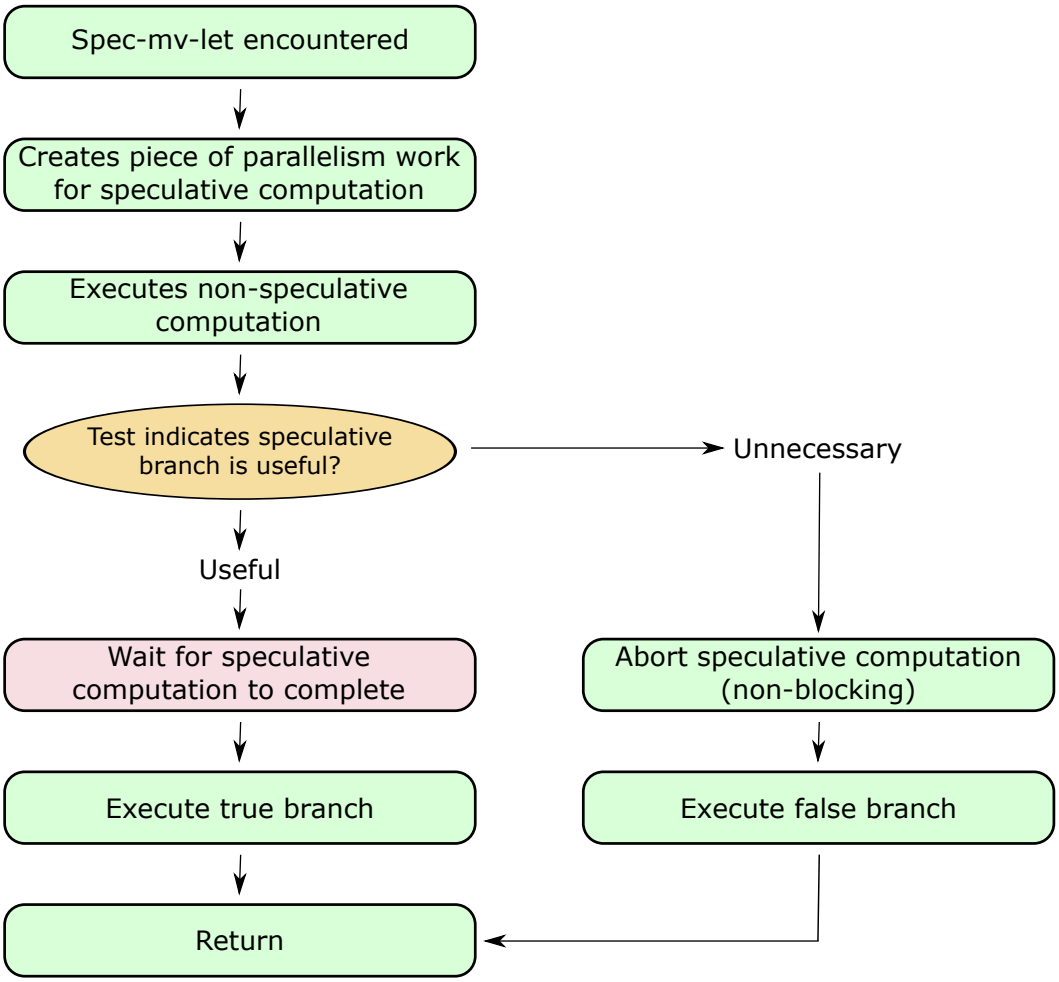

Figure 4.7: Life of a `Spec-mv-let`

in Figure 4.6. First, `<spec>` is executed speculatively (as our implementation of `spec-mv-let` wraps `<spec>` inside a call of `future`). Control then passes immediately to the `mv-let` call, without waiting for the result of executing `<spec>`. The variables `(w1 ... wk)` are bound to the result of executing `<eager>`, and then `<test>` is executed (note that it is an error to reference `(v1 ... vn)` within `<test>`). If the computed value of `<test>` is true, then the values of `(v1 ... vn)` are needed, and `<typical-case>` blocks until they are available. As such, the `<typical-case>` may make reference to the variables bound as part of the speculative execution, `(v1 ... vn)`. If the value of `<test>` is `nil`, then the values of `(v1 ... vn)` are assumed to be unnecessary, and the execution of `<spec>` may be aborted. As a result, the variables `(v1 ... vn)` may not be referenced by `<abort-case>`.

A graphical view of `spec-mv-let`'s flow can be found in Figure 4.7. When the `spec-mv-let` primitive is encountered, it creates a piece of parallelism work that represents the computation to perform speculatively (labeled `<spec>` above). This piece of parallelism work is executed by our parallelism execution engine, described in Chapter 8 (the reader may be particularly interested in diagrams that describe what happens to a piece of parallelism work, shown in Figure 8.1, and how we use threads to execute pieces of parallelism work, shown in Figure 8.2). The non-speculative computation (`<eager>`) is then executed in the current thread. After the non-speculative computation finishes, the `<test>` is executed. If the test indicates that the speculative computation is useful (by returning a non-`nil` value), the current thread blocks

until the speculative computation finishes. After the speculative computation finishes, `spec-mv-let` returns the value computed by the true branch of the test. On the other hand, if the test indicates that the speculative computation is unnecessary (by returning `nil`), the speculative computation is aborted, and `spec-mv-let` returns the value that the false branch computes.

4.3 Performance Results for Futures and `Spec-mv-let`

In this section we present two types of performance results. First, we provide evidence that *the overhead of using a future is approximately 45 microseconds, and that the overhead for using spec-mv-let is approximately 45 or 33 microseconds*, depending upon whether the speculative `mv-let` binding is used. The second set of performance results use futures and `spec-mv-let` in naïve definitions of the Fibonacci function, comparing their times for parallel and serial executions. See *Implementing a Parallelism Library for a Functional Subset of Lisp* [45] for a similar assessment of the performance of `plet`, `pargs`, `pand`, and `por`.

4.3.1 Overhead of a Future

Here we include a test and its results that indicate how long it takes to create a future, which will be executed in another thread, and then read its resulting value from the original thread. We submit the two forms shown in Figure 4.8 to the Lisp prompt of ACL2. The shown script requires 44.52 seconds to complete (on machine *older-8-core-nht*, which is detailed in Sec-

```
(defun make-and-read-future ()
  (future-read (future 3)))
(time$ (dotimes (i 1000000)
  (make-and-read-future)))
```

Figure 4.8: Script to Determine the Overhead of a Future

tion 5.2.1), so the overhead for creating, executing, and reading a future is approximately 45 microseconds.

Thus, if the length of time it takes to execute a subgoal in ACL2(p) is typically much more than 45 microseconds, futures will be efficient enough to provide the basic primitive for parallelizing our application. Our futures implementation also provides an efficient mechanism for aborting computation. By running the test shown in Figure 4.9, we can see how long it takes to abort computation that has already been added to the parallelism work queue. The script shown in Figure 4.9 requires approximately 75 seconds to finish. Thus, *it takes about 75 microseconds to spawn a future and then completely abort its execution.*

In the script shown in Figure 4.9, we call function `count-down`, which is designed to consume CPU time (`(count-down 1,000,000,000)` typically requires about 5 seconds). Since calling `mistake` 1,000,000 times only requires approximately 75 seconds, and since we wait for the number of futures in the system (`*total-future-count*`) to be zero, we know that we are waiting for the computation to completely finish aborting. This extra check is important because `future-abort` itself sets a flag and if a thread is executing the future, `future-abort` interrupts that thread, and tells it to abort.

However, `future-abort` does not block and wait for that thread to unwind and actually stop executing the future. Given our implementation, waiting for `*total-future-count*` to be zero ensures that the thread executing the future is finished executing.

```
(defun count-down (x)
  (declare (xargs :guard (natp x)))
  (cond ((zp x)
         0)
        (t
         (count-down (1- x)))))

(defun mistake ()
  (future-abort (future (count-down 1000000000)))
  (let ((required-waiting nil))
    (loop while (> *total-future-count* 0)
          do
          (incf *waiting-cycles*)
          (setq required-waiting t))
    (if required-waiting
        (incf *futures-that-were-not-done-when-abort-issued*)
      (incf *futures-completed-before-abort-issued*)))
  t)

(time$-with-gc-and-recording
  '|1,000,000-mistakes|
  (dotimes$ (i 1000000)
            (mistake)))
```

Figure 4.9: Script to Determine the Amount of Time Required to Spawn and Completely Abort a Future

4.3.2 Overhead of Spec-mv-let

In this section we determine the overhead of using `spec-mv-let`. We do this by executing two types of tests. The first test, shown in Figure 4.10, requires 44.77 seconds to complete. Thus, it takes approximately 45 microseconds to complete the evaluation of `spec-mv-let` when taking the branch that

```
(defun always-valid-speculation ()
  (spec-mv-let (x y)
               (mv 3 4)
               (mv-let (q r)
                       (mv 7 8)
                       (if t
                           (+ x y q r)
                         "wrong!"))))
(time$ (dotimes (i 1000000)
         (always-valid-speculation)))
```

Figure 4.10: Script to Determine Overhead of `Spec-mv-let` When Test Is Valid

```
(defun always-invalid-speculation ()
  (spec-mv-let (x y)
               (mv 3 4)
               (mv-let (q r)
                       (mv 7 8)
                       (if nil
                           (or "wrong!" (+ x y))
                         (+ q r)))))
(time$ (dotimes (i 1000000)
        (always-invalid-speculation)))
```

Figure 4.11: Script to Determine Overhead of `Spec-mv-let` When Test Is Invalid

uses the speculative execution result. The second test, shown in Figure 4.11, requires 33.48 seconds to complete. Thus, it takes approximately 33 microseconds to complete the evaluation of `spec-mv-let` when the test returns `nil` and the speculative branch is aborted.

4.3.3 Using Futures and Spec-mv-let with Fibonacci

In this section we test the performance of futures and `spec-mv-let` by defining naïve definitions of the doubly recursive Fibonacci function. All testing was performed on the eight-core 64-bit Linux machine *older-8-core-nht* (see Section 5.2.1 for further details of this machine) running 64-bit CCL

with the Ephemeral Garbage Collector (EGC) disabled and a two gigabyte Garbage Collection (GC) threshold. We also check that the garbage collector does not run during the execution of each test. All times are reported in seconds, and each reported speedup factor is a ratio of serial execution time to parallel execution time. In each case, we report minimum, maximum, and average times for ten consecutive runs of each test, both parallel and serial, in the same environment. Recall the definition of `pfib` in Figure 4.2. In our experiments, calling `(pfib 45)` yields a speedup factor of 7.43x on an eight-core machine (performance times shown in Figure 4.13).

We next define a parallel version of the Fibonacci function inside the ACL2 logic by using `spec-mv-let`. The support for speculative execution provided by `spec-mv-let` is unnecessary here, since we always need the result of both recursive calls; but our purpose here is only to benchmark `spec-mv-let`. As shown in Figure 4.13, the following definition has provided a speedup factor of 7.38x when executing `(pfib 45)`. From these results, we conclude that `spec-mv-let` provides a useful amount of speedup for our parallelized definition of the Fibonacci function. This is an indicator that `spec-mv-let` may also be able to provide a useful amount of speedup when used inside the ACL2 waterfall.

```
(defun pfib (x)
  (declare (xargs :guard (natp x)))
  (cond ((or (zp x) (<= x 33))
         (fib x))
        (t (spec-mv-let (fib-x-1)
                        (pfib (- x 1))
                        (mv?-let (fib-x-2)
                                 (pfib (- x 2))
                                 (if t
                                     (+ fib-x-1 fib-x-2)
                                   "Unreachable branch"))))))
```

Figure 4.12: Definition of Fibonacci Using `Spec-mv-let`

Case	Min	Max	Avg	Speedup
Serial	53.062	53.069	53.066	
Futures	7.066	7.224	7.139	7.43x
Spec-mv-let	7.128	7.395	7.195	7.38x

Figure 4.13: Performance of Parallelism Primitives in the Fibonacci Function (times reported in seconds)

Chapter 5

Underlying Runtime Performance Characteristics

The performance of the underlying runtime system varies with every machine configuration. In this section, we examine the performance of our test machines by running code that performs simple tasks like counting down from a large number or checking that each element of an array has the correct value. We run our tests on three machines: (1) an eight-core machine that contains older processors with no support for hyper-threading, (2) a four-core machine that contains modern processors with two-way hyper-threading, and (3) a twenty-core machine that contains modern processors with two-way hyper-threading.

Other implementors of parallel systems may wish to run similar tests to determine the performance capabilities of their underlying runtime environment. Be forewarned that the tests need to be long enough; i.e., our initial results for Section 5.2.3 were non-trivially different until we lengthened the duration of the tests by a factor of four.

5.1 Introducing the Test Scripts

Here we present two sets of test scripts. The first script counts down from a very large number in either a single thread or multiple threads. This test is intended to represent functions that do not access much memory. The second script checks that every element in an array contains the correct value, also in either a single thread or multiple threads. This test is intended to represent functions that access memory (in practice, the cache lines). Note that these test scripts are not intended to be comprehensive; we only run them to obtain background information on our test machines.

5.1.1 Counting Down

We define the function `count-down` to just count down to zero, decrementing its argument by one with each recursion. Here is its definition:

```
(defun count-down (x)
  (cond ((zp x)
         0)
        (t
         (count-down (1- x)))))
```

We then define a function named `count-down-with-signal` that signals a semaphore after `count-down` finishes counting down from a given number.

```
(defun count-down-with-signal (x)
  (count-down x)
  (signal-semaphore *done*))
```

We then define another function, `count-down-for-tests`, which takes as arguments `duration-type`, which indicates whether we should run the short

or long version of the test, and `threading-type`, which indicates whether we should run it in a single thread or with multiple threads:

```
(defun count-down-for-tests (duration-type threading-type)
  (let ((duration (if (equal duration-type :short)
                      *small-number-for-counting*
                    *large-number-for-counting*)))
    (cond ((equal threading-type :single)
           (dotimes (i *core-count*)
             (count-down-with-signal duration))
           (dotimes (i *core-count*)
             (wait-on-semaphore *done*)))
          (t ; :multi case
           (dotimes (i *core-count*)
             (run-thread "test-thread"
                         #'count-down-with-signal
                         duration))
           (dotimes (i *core-count*)
             (wait-on-semaphore *done*)))))))
```

5.1.2 Looping Through an Array

We define the function `array-loop` to loop through an array and check that each slot in the array is equal to a particular value:

```
(defun array-loop (duration)
  (dotimes (i (round (/ duration 1600)))
    (dotimes (j (expt 2 10))
      (assert (equal (aref *my-array* j) 'hi)))))
```

We then define a function named `array-loop-with-signal` to signal a semaphore after `array-loop` finishes looping through the array:

```
(defun array-loop-with-signal (duration)
  (array-loop duration)
```

```
    (signal-semaphore *done*))
```

We then define another function, `array-loop-for-tests`, which takes as arguments `duration-type`, which indicates whether we should run the short or long version of the test, and `threading-type`, which indicates whether we should run it in a single thread or with multiple threads.

```
(defun array-loop-for-tests (duration-type threading-type)
  (let ((duration (if (equal duration-type :short)
                      *small-number-for-counting*
                    *large-number-for-counting*)))
    (cond ((equal threading-type :single)
           (dotimes (i *core-count*)
             (array-loop-with-signal duration))
           (dotimes (i *core-count*)
             (wait-on-semaphore *done*)))
          (t ; :multi case
           (dotimes (i *core-count*)
             (run-thread "test-thread"
                         #'array-loop-with-signal
                         duration))
           (dotimes (i *core-count*)
             (wait-on-semaphore *done*))))))
```

5.1.3 Running the Tests

We now define a macro, `run-and-record`, which takes three arguments. The argument `key` is a key to use when saving the time it takes to execute the argument `form`, and the argument `iterations` indicates how many times the test should be executed.

```
(defmacro run-and-record (key form iterations)
  `(dotimes$
```

```
    (i ,iterations)
    (time$-with-gc-and-recording (quote ,key) ,form)))
```

The following macro, `time$-with-gc-and-recording`, accepts two arguments, a `key` under which timing information will be saved and a `form` to execute. This macro also runs the (single-threaded) garbage collector before each test and asserts that it does not run during the test. Thus, any variations in performance will not be due to garbage collection.

```
(defmacro time$-with-gc-and-recording (key form)
  `(let* ((ignored1
           (progn (format t "Running gc before starting ~
                             form~%")
                  (gc$)
                  (format t "Starting form~%")))
          (start-gc-time (gc-run-time))
          (start-wall-time (get-internal-real-time))
          (result (multiple-value-list (time$ ,form)))
          (end-gc-time (gc-run-time))
          (end-wall-time (get-internal-real-time))
          (total-gc-time (- end-gc-time start-gc-time))
          (total-wall-time (- end-wall-time start-wall-time)))
     (declare (ignore ignored1))
     (assert (equal total-gc-time 0))
     (format t "Total GC time (in micro-sec): ~s~%"
             total-gc-time)
     (format t "Total wall time (in micro-sec: ~s~%"
             total-wall-time)
     (record-timing ,key total-wall-time)
     (values-list result)))
```

We also define the constant `*iterations*`, which we use as the number of times to run each test. Statistically speaking, `*iterations*` is our sample

size.

```
(defconst *iterations* 10)
```

Thus, calling the following form results in calling function `count-down` ten times, with a duration that results in a short version of the test, in a single thread, and saving the amount of time it takes to run that test under the key `count-down-short-single`.

```
(run-and-record count-down-short-single
                (count-down-for-tests :short :single)
                *iterations*)
```

We also have a function named `use-hyperthreading` that configures the number of threads to use in the multi-threaded test case. When we call `use-hyperthreading` with a value of `t`, when the test runs, it will use a number of threads equal to the number of hardware threads supported by the underlying machine. The following definitions constitute our implementation of this function.

```
(defconst *host-to-core-count-map*
  '(("lhug-7" . 8)
    ("dunnottar" . 4)
    ("glamdring-4" . 20)))

(defconst *host-to-hardware-thread-count-map*
  '(("lhug-7" . 8)
    ("dunnottar" . 8)
    ("glamdring-4" . 40)))

(defun use-hyperthreading (use-hyperthreading)
  (setf *reset-core-count-too* nil)
  (mv-let
   (erp hostname state)
   (getenv$ "RAGER_HOSTNAME" *the-live-state*)
```

```
(declare (ignore erp state))
(cond (use-hyperthreading
       (setf *core-count*
             (cdr (assoc-equal
                   hostname
                   *host-to-hardware-thread-count-map*))))
      (t (assert (null use-hyperthreading))
         (setf *core-count*
               (cdr (assoc-equal
                     hostname
                     *host-to-core-count-map*)))))
(when (null *core-count*)
  (er hard 'use-hyperthreading
      "Either the thread-count or core-count map is missing
       an entry for hostname ~x0"
      hostname))))
```

We also define a function, `print-timings-latex`, that prints the timing results for the most recent set of tests. This function also prints whether hyper-threading was enabled for those tests.

Given all of these definitions, we are able to run the script shown in Figure 5.1 to obtain results for counting down and looping through an array for both a single thread and multiple threads. A similar script exists for the hyper-threaded case.

5.2 Performance Results

We now show the results of running the above described tests on three different machines, which we refer to as: *older-8-core-nht*, *modern-4-core-2ht*, and *modern-20-core-2ht*. Included in each section are the specification of the

```
(use-hyperthreading nil)
(run-and-record count-down-short-multi
                (count-down-for-tests :short :multi)
                *iterations*)
(run-and-record count-down-short-single
                (count-down-for-tests :short :single)
                *iterations*)
(run-and-record count-down-long-multi
                (count-down-for-tests :long :multi)
                *iterations*)
(run-and-record count-down-long-single
                (count-down-for-tests :long :single)
                *iterations*)
(run-and-record array-loop-short-multi
                (array-loop-for-tests :short :multi)
                *iterations*)
(run-and-record array-loop-short-single
                (array-loop-for-tests :short :single)
                *iterations*)
(run-and-record array-loop-long-multi
                (array-loop-for-tests :long :multi)
                *iterations*)
(run-and-record array-loop-long-single
                (array-loop-for-tests :long :single)
                *iterations*)
```

Figure 5.1: Script to Run `Count-down` and `Array-Loop` Tests with Hyperthreading Disabled and Record Their Performance Results

machine, the minimum, maximum, and average times required to run each test across ten iterations, and a discussion concerning how the results affect our work. Each machine implements the 64-bit x86 architecture.

The column labeled "Su-Avg" is the speedup determined by dividing the *average* time it takes the single-threaded test to finish by the average time it takes the multi-threaded test to finish. The column labeled "Su-Min"is the speedup determined by dividing the *minimum* time it takes the single-threaded test to finish by the minimum time it takes the multi-threaded test to finish. Given we want to examine "best case" performance, it is probably best to examine the speedup as determined with the minimum times. However, we also include the speedup for the average times to more accurately represent the overall performance of the underlying runtime system. These tests are intended to establish "best-case" speedup for each machine, and the underlying constants that we used for each set of tests are different for each machine. Therefore, the reported times (either single-threaded or multi-threaded) should not be compared between machines.

5.2.1 Performance of an Older Multi-Core Machine

Our first test machine is *older-8-core-nht*. This machine has the following specifications:

DNS Name: lhug-7.cs.utexas.edu
Processors: 4
Number of Cores Per Processor: 2
Total Number of Cores: 8

Total Number of Hardware Threads: 8
Memory: 32 gigabytes
Processor Model Name: AMD Opteron (tm) Processor 850 @ 1.8GHz

Figure 5.2 shows the performance results for running the tests on *older-8-core-nht*. From these results, we conclude that when we execute large enough pieces of work in parallel, that we can obtain close to linear speedup on *older-8-core-nht*.

Test	Duration	Threading	Avg	Su-Avg	Min	Su-Min
Count-down	Short	Multi	0.180		0.163	
Count-down	Short	Single	1.241	6.894	1.241	7.613
Count-down	Long	Mult	3.123		3.106	
Count-down	Long	Single	24.814	7.946	24.809	7.987
Array-loop	Short	Multi	0.180		0.161	
Array-loop	Short	Single	1.221	6.783	1.221	7.584
Array-loop	Long	Multi	3.072		3.064	
Array-loop	Long	Single	24.433	7.953	24.414	7.968

Figure 5.2: Performance Results for Eight Threads on *Older-8-core-nht*

5.2.2 Performance of a Modern Machine with Four CPU Cores

Our second test machine is *modern-4-core-2ht*. This machine has the following specifications:

DNS Name: dunnottar.cs.utexas.edu
Processors: 1
Number of Cores Per Processor: 4
Total Number of Cores: 4
Total Number of Hardware Threads: 8
Memory: 32 gigabytes

Processor Model Name: Intel(R) Xeon(R) CPU E31280 @ 3.50GHz

Figure 5.3 shows the results of running the tests on *modern-4-core-2ht* with four threads. Figure 5.4 shows the results of running the tests on *modern-4-core-2ht* with eight threads.

Test	Duration	Threading	Avg	Su-Avg	Min	Su-Min
Count-down	Short	Multi	0.297		0.296	
Count-down	Short	Single	1.071	3.606	1.070	3.614
Count-down	Long	Mult	5.790		5.790	
Count-down	Long	Single	21.425	3.700	21.423	3.700
Array-loop	Short	Multi	0.266		0.265	
Array-loop	Short	Single	0.952	3.579	0.952	3.592
Array-loop	Long	Multi	5.165		5.164	
Array-loop	Long	Single	19.040	3.686	19.038	3.687

Figure 5.3: Performance Results for Four Threads on *Modern-4-core-2ht*

Test	Duration	Threading	Avg	Su-Avg	Min	Su-Min
Count-down	Short	Multi	0.491		0.491	
Count-down	Short	Single	2.146	4.371	2.145	4.369
Count-down	Long	Mult	9.786		9.785	
Count-down	Long	Single	42.896	4.383	42.854	4.380
Array-loop	Short	Multi	0.444		0.443	
Array-loop	Short	Single	1.907	4.295	1.907	4.305
Array-loop	Long	Multi	8.848		8.844	
Array-loop	Long	Single	38.127	4.309	38.122	4.310

Figure 5.4: Performance Results for Eight Threads on *Modern-4-core-2ht*

In these results we start to see some issues with scaling, even for code designed to scale linearly with respect to the number of CPU cores. On a

positive note, we observe about an 18% improvement in performance when taking advantage of hyper-threading. Thus, we see that hyper-threading can be beneficial but not nearly as helpful as an additional core.

5.2.3 Performance of a Modern Machine with Twenty CPU Cores

Our third test machine is *modern-20-core-2ht*. This machine has the following specifications:

DNS Name: glamdring-4.cs.utexas.edu
Processors: 2
Number of Cores Per Processor: 10
Total Number of Cores: 20
Total Number of Hardware Threads: 40
Memory: 512 gigabytes
Processor Model Name: Intel(R) Xeon(R) CPU E7- 2870 @ 2.40GHz

For all of our tests on *modern-20-core-2ht*, we disabled Intel's Turbo Speedstep 2.0 feature [21] at the BIOS level. Even with this feature enabled in the BIOS, we found the results to be very similar. Thus, to reduce the amount of information that the reader must parse in order to understand our work, we omit those results from this dissertation.

Figure 5.5 shows the results of running the tests on *modern-20-core-2ht* with twenty threads. Figure 5.6 shows the results of running the tests on *modern-20-core-2ht* with forty threads. From these results, we can see that *modern-20-core-2ht* scales slightly better than *modern-4-core-2ht* on both simple tests. We also note that we gain about 5% performance by using hyper-

threading in the count-down test, and that we actually incur a loss in the array-loop test. These are smaller gains than achieved on *modern-4-core-2ht*.

Test	Duration	Threading	Avg	Su-Avg	Min	Su-Min
Count-down	Short	Multi	0.747		0.607	
Count-down	Short	Single	8.710	11.659	8.697	14.327
Count-down	Long	Mult	9.050		8.831	
Count-down	Long	Single	173.947	19.221	173.942	19.697
Array-loop	Short	Multi	0.776		0.602	
Array-loop	Short	Single	8.589	8.836	8.580	14.252
Array-loop	Long	Multi	9.156		9.003	
Array-loop	Long	Single	171.585	18.740	171.578	19.058

Figure 5.5: Performance Results for Twenty Threads on *Modern-20-core-2ht*

Test	Duration	Threading	Avg	Su-Avg	Min	Su-Min
Count-down	Short	Multi	1.251		1.128	
Count-down	Short	Single	17.414	13.920	17.393	15.419
Count-down	Long	Mult	16.856		16.782	
Count-down	Long	Single	347.862	20.637	347.852	20.728
Array-loop	Short	Multi	1.284		1.240	
Array-loop	Short	Single	17.172	13.374	17.159	13.838
Array-loop	Long	Multi	18.355		18.233	
Array-loop	Long	Single	343.167	18.696	343.150	18.820

Figure 5.6: Performance Results for Forty Threads on *Modern-20-core-2ht*

Chapter 6

Interactive Issues in Managing Parallel Execution

In this chapter we cover the user-level issues related to managing the parallel execution of the main ACL2 proof process (named "the waterfall"), modification of the program state from within the waterfall, and proof output in ACL2(p).

6.1 Enabling Waterfall Parallelism

The main method for enabling waterfall parallelism is an ACL2 macro `set-waterfall-parallelism`. `Set-waterfall-parallelism` accepts one argument that specifies under what conditions the waterfall should parallelize execution. The possible arguments are `nil`, `:full`, `:top-level`, `:resource-based`, `:resource-and-timing-based`, and `:pseudo-parallel`. *Resource-based is the recommended setting for ACL2(p).* We next outline the defining characteristics of each setting.

Nil A value of `nil` indicates that ACL2(p) should never prove subgoals in parallel. This setting causes ACL2(p) to behave just like ACL2, which allows inherently single-threaded code to be used. For instance, this setting permits

users to use *hints* that modify global variables inside the waterfall (see Section 6.3 for further details concerning the use of such hints).

Full A value of `:full` indicates that ACL2(p) should always prove independent subgoals in parallel. We impose a limit on the total amount of parallelism work allowed in the system, even for `full` mode. See Section 6.2 for a guide to the user-level mechanisms that adjust this limit. Before we created this limit, it was possible to cause the Linux daemon Watchdog [35] to reboot a machine by attempting a proof with tens of thousands of subgoals. See Section 8.3.2.3 for implementation-level details of this problem.

Top Level A value of `:top-level` indicates that ACL2(p) should prove each of the top-level subgoals (before induction) in parallel but otherwise prove subgoals in a serial manner. For example, we thought this mode could be useful when the users know that there are enough top-level subgoals, many of which take a non-trivial amount of time to prove, such that proving them in parallel would result in a useful reduction in overall proof time. However, after benchmarking the different modes, we observed that `top-level` never outperforms `full` or `resource-based` waterfall parallelism, and in most cases, it performs worse. As such, it should be considered only for experimental use.

Resource Based A value of `:resource-based` indicates that ACL2(p) should use its built-in heuristics to determine whether parallelism resources

are available for parallel execution. Note that ACL2(p) does not hook into the operating system (OS) to determine the current workload of the machine. ACL2(p) assumes that it is the only process using significant CPU resources, and it optimizes the amount of parallel execution based on the number of CPU cores in the system and the total number of threads supported by the underlying runtime system. *Resource-based is the recommended setting for waterfall parallelism.* See Section 8.3 and its subsections for a detailed explanation of how ACL2(p) determines whether multi-threading resources are available and limitations of this mode.

Resource and Timing Based During the first proof attempt of a given conjecture, a value of `:resource-and-timing-based` results in the same behavior as would result with the `resource-based` setting. However, on subsequent proof attempts, the time it took to prove each subgoal will be considered when deciding whether to parallelize execution. If a particular theorem's proof is already achieving satisfactory speedup via `resource-based` parallelism, there is no reason to try this setting. We initially thought that the `resource-and-timing-based` setting could improve performance of subsequent proof attempts (see the following paragraphs for why we changed our minds). Note that since the initial run does not have the subgoal proof times available, an initial proof attempt will never perform better under this mode than the `resource-based` mode. Also note that `resource-and-timing-based` will never perform better than the `resource-based` mode in a non-interactive

session where each proof is only performed once.

We did not further pursue the implementation of the `resource-and--timing-based` mode for one main reason. The initial idea behind our implementation was only to parallelize the proofs of subgoals that met a particular granularity threshold. As explained in Section 8.3.1, we discovered that the granularity problem is not an issue when parallelizing the proofs of ACL2 subgoals. Therefore, the idea of avoiding overhead for parallelizing subgoals that are too short-lived to be worth parallelizing turned out to be unnecessary.

A less important secondary reason we did not further pursue this mode is that we needed a reliable "key" for each subgoal when storing the timing information. It is well-known among the ACL2 community that finding such a "key" can be tricky. The simplest thing is to use the subgoal number (e.g., *Subgoal 1.1.7*) as the key, and we do that in our current implementation of `resource-and-timing-based`. This strategy has the weakness that if a proof attempt's path changes slightly, the recorded timing information will no longer be correctly matched, rendering the recorded timing information not only useless but misleading. We could fix this by using a hash of a subgoal's formula as the key, which would be more resistant to changes in the proof's path. However, users could still alter the proof attempt (e.g., by enabling and disabling rewrite rules), possibly changing the duration of any particular subgoal's proof attempt, rendering the mode, once again, less useful.

As mentioned in Section 7.2.4, we could imagine a mode for ACL2(p) that prioritized the proof attempts of the subgoals along the critical path of

a proof. If we were to rework the underlying futures library to provide a way to prioritize some futures over other futures, then it would be useful to store timing information for each subgoal under its appropriate key and use that information to prioritize. We leave this as future work.

Pseudo Parallel A value of `:pseudo-parallel` results in using the parallel version of the theorem proving code, but with serial execution. This setting is useful for two reasons. First, this mode is particularly useful if the user is a developer who would like to see comprehensible output from tracing the parallel version of the proof process. If the reader is parallelizing a non-trivial application, providing a pseudo-parallel configuration should be considered so that debugging is easier. Second, we use this mode to determine a relevant serial proof duration when computing the speedup of different waterfall parallelism modes. In practice, since all of the waterfall parallelism modes omit some of the code for functionality like output,the `pseudo-parallel` mode takes slightly less time than the serial mode (represented with a `nil` value given to `set-waterfall-parallelism`). As such, using the `pseudo-parallel` mode as the baseline for calculating speedup is more fair than using the time it takes with a waterfall parallelism setting of `nil` (it is also more fair than comparing the parallel execution timing results to the version of ACL2 that does not support parallel execution at all).

6.2 Configuring ACL2(p) for When Too Much Parallelism Work is Encountered under `Full` Waterfall Parallelism

As mentioned in Section 6.1, it was once possible to exhaust the underlying machine resources when using the `full` waterfall parallelism mode. As will be discussed in Section 8.3.2.3, our solution to this problem is to implement a limit on the total amount of parallelism work allowed in the system and cause a relatively clean ACL2 error when this limit is about to be exceeded. This prevents users from crashing their machines, but what if they wish to incur such a risk and manage the limit themselves? This ability to let users manage the limit is important because each user's platform has different limitations. In this section, we introduce two functions: `set-total-parallelism-work-limit-error`, which allows users to specify what happens when the limit is reached, and `set-total-parallelism-limit`, which allows users to set the threshold that triggers the error.

By default, when the total amount of parallelism work in the system reaches the limitations of the underlying runtime system, the ACL2(p) user receives an error and computation halts. At this point, the ACL2(p) user has the following three options:

- Disable the error so that execution continues serially whenever the underlying multi-threading resources are exhausted. The following form accomplishes this:

  ```
  (set-total-parallelism-work-limit-error nil)
  ```

- In spite of the potential risk, increase the limit on the amount of parallelism work that ACL2(p) is willing to create. In this case, the user can obtain the current limit by issuing the following query:

  ```
  (f-get-global 'total-parallelism-work-limit state)
  ```

  Then to increase that limit, the user can execute the following form:

  ```
  (set-total-parallelism-work-limit <new-integer-value>)
  ```

  For example, suppose that the value of `total-parallelism-work-limit` was originally 8,000 and the user wishes to increase that limit to 13,000. Execution of the following form performs such an increase:

  ```
  (set-total-parallelism-work-limit 13000)
  ```

- Completely remove the use of the limit by submitting the following form:

  ```
  (set-total-parallelism-work-limit :none)
  ```

We now include a specification for `set-total-parallelism-work--limit-error` and `set-total-parallelism-work-limit`.

Set-total-parallelism-work-limit-error Users can control the action taken when the underlying multi-threading resources are exceeded by calling function `set-total-parallelism-work-limit-error` with an argument of `t` or `nil`. A value of `t` (the default setting for ACL2(p)) indicates that an error should occur, prompting users either to pick a new total parallelism work limit or to change the behavior that occurs when encountering the limit. A value of `nil` indicates that no error should occur, and assuming `set-total-parallelism-work-limit` has not been called with a value of

`:none`, execution will continue serially until the total amount of parallelism work in the system falls below the limit.

Set-total-parallelism-work-limit The function `set-total-parallelism--work-limit` sets the limit on the total amount of parallelism work allowed into the system. While the system is at this limit, parallelism primitives execute serially until the total amount of parallelism work in the system decreases below the limit.

ACL2 initially uses a conservative estimate to limit the amount of parallelism work allowed into the system. To tell ACL2(p) to use a different limit, users can call `set-total-parallelism-work-limit`, a function whose single argument is either the value `:none` or an integer that represents the new threshold. Passing in the value `:none` represents passing in a value of infinity, and ACL2(p) will always continue to parallelize the waterfall under the `full` setting, even when ill-advised. Passing in an integer value simply replaces the previous limit with the new one.

The default limit on the total amount of parallelism work, currently 8,000 pieces, is also accessible to users by calling function `default-total--parallelism-work-limit`.

6.3 Managing the Modification of ACL2's Global State from within the Waterfall

In our first attempt at removing the modification of *state* (see Section 3.2) from the waterfall, we caused a relatively uninformative error any time *state* was about to be returned. This resulted in an error for approximately 7% of the ACL2 regression suite. After this initial attempt, we decided to examine the modification of the program state in three main areas: (1) proof *hints*, (2) the mechanism that translates user-level terms into their internal representation, and (3) the evaluation mechanism used within the waterfall. Since this chapter focuses on interactive issues, we discuss (1) in the following section and leave (2) and (3) for discussion in Chapter 9.

6.3.1 ACL2 Mechanism for Using Proof Hints that Can Modify State

Many of the ACL2 prover functions that returned *state* were part of ACL2's *hint* mechanism, which gives users a means to guide the theorem proving process. Some of these hints are processed before entering the waterfall, but many of them are processed while the waterfall is already executing. Since we execute the waterfall in parallel, hints that modify the global program state are potentially problematic. However, not all uses of hints that are allowed to modify *state* cause problems in practice. For example, such hints might only modify *state* to perform printing, and as long as this printing is done using the ACL2 output lock (see "with-<*lock-name*>" in Section 4.1.3), it may be perfectly reasonable to allow this printing to occur. For cases like this,

we eventually provided a mechanism that allows users to use single-threaded hints that modify *state*, and this mechanism is described in Section 6.3.2.

In our first attempt at handling hints that were inherently not safe for parallel execution, we setup ACL2 so that these hints would work silently, without warning users that they were doing something inherently single-threaded in a program running with parallel execution. Most of the regression suite was able to pass under this configuration. However, we were not protecting users from the inherently dangerous behavior of executing code that modified the global program state in parallel – omitting such protection would have been a major usability issue. Therefore, we next disallowed the modification of the program state from inside the waterfall with ACL2 hint mechanisms, this time providing detailed error messages that included the offending forms and further details that helped users learn how to correct their problem. In our current and final version, we give users the ability to override this error and continue anyway. The remainder of this section explains our implementation.

6.3.2 ACL2 Mechanism for Letting Users Run Single-threaded Hints in Parallel

We now detail the ACL2(p) interface that permits the use of inherently single-threaded hints when executing the waterfall in parallel. To understand our solution, one must first understand the pre-existing notion of an ACL2 *trust tag* (see documentation topic "defttag" inside the ACL2 manual [1]). Trust tags allow users to do things that they believe to be sound but are not

certified as sound by the ACL2 system. As such, trust tags serve as red flags that a proof script contains something potentially risky. ACL2(p) requires that users define trust tags before using hints that modify the global program state while waterfall parallelism is enabled. This is a reasonable requirement, because we expect ACL2(p) to be used interactively. So, while users might opt to define trust tags while developing their proofs, they would not need it for their final certifications, which would occur serially.

Set-waterfall-parallelism-hacks-enabled To enable the use of hints that modify the program state, users can call macro `set-waterfall-parallelism--hacks-enabled` with an argument of `t`. Similarly, to disable the use of such hints, users can call the same macro with an argument of `nil`. Calling this macro with a value of `t` requires that a trust tag be pre-defined. If a user wishes to have ACL2(p) automatically define a trust tag, then the user can call `set-waterfall-parallelism-hacks-enabled!` with an argument of `t`, which automatically declares such a trust tag.

6.4 Managing Proof Output

In ACL2(p), since the proofs of many subgoals can be attempted in parallel, proof checkpoints (introduced in Section 3.3) can be available for printing significantly sooner than in the serial version of the theorem prover. For printing output, ACL2(p) follows the precedent of a mode in the non-parallel version of ACL2 called *gag mode* (see documentation topic "gag-mode"

in the ACL2 Manual [1]). The idea behind gag mode is to print only the subgoals most relevant to debugging a failed proof attempt. These subgoals are called *key checkpoints*, and we restrict the default output mode for the parallel version of the waterfall to printing checkpoints similar to these key checkpoints (the parallel version of the waterfall prints key checkpoints that are "unproved" in the following sense: a subgoal is a key checkpoint if it leads, in the current call of the waterfall, to a goal that is later pushed for induction"). Section 6.4.1 explains the options available to users for controlling the output provided when executing the waterfall in parallel.

6.4.1 ACL2 Mechanisms for Controlling Proof Output

The macro that configures the printing that occurs inside the waterfall is `set-waterfall-printing`. The following three options for calling this macro are described below. *A setting of very-limited, along with enabling gag mode (perhaps with a call of* `(set-gag-mode t)`*, to suppress some of the output that occurs outside the waterfall), is the recommended setting for ACL2(p).*

Full A value of `:full` is intended to print output that is the same as the output that occurs in the non-parallel version of ACL2. This output will be interleaved and typically unreadable unless the waterfall parallelism mode is one of `nil` or `pseudo-parallel`.

Limited A value of `:limited` omits most of the output that occurs in the serial version of the waterfall. Instead, the proof attempt prints proof

checkpoints, similar to gag mode. The `limited` mode also prints messages that indicate which subgoal is currently being proved, along with the wall-clock time elapsed since the theorem began its proof; and if global variable `waterfall-printing-when-finished` has a non-`nil` value, then such a message will also be printed at the completion of each subgoal. Naturally, these subgoal numbers can appear out of order, because the subgoals are being proved in parallel.

Very Limited A value of `:very-limited` is treated the same as `:limited`, except that instead of printing subgoal numbers and timing information, the proof attempt prints a period (".") each time it starts a new subgoal. Also, if global variable `waterfall-printing-when-finished` is set to a non-`nil` value, a comma (",") is printed every time a subgoal finishes.

6.4.2 Implementation Note on Printing Proof Checkpoints

As discussed in Section 6.3, most ACL2 functions that have side-effects, like output, must accept *state* and return *state* as part of their return value. Indeed, in the serial version of the waterfall, the output that occurs during the waterfall's execution is performed by functions with this property. However, in the parallel version of the waterfall, we cannot pass around a modified version of *state* – ACL2 prevents us syntactically from doing so. As such, after incorporating the necessary locks to guarantee mutual exclusion, we perform the output using a form of printing called *comment window* printing (see doc-

umentation topic "cw" inside the ACL2 manual [1]), which does not accept or return *state*. This allows us to still print the output relevant to debugging a proof while also removing the modification of *state* from the waterfall.

Chapter 7

Proof Parallelism Potential and Results

We enumerate four types of proof attempts, categorized with respect to their ability to realize the benefits from parallel execution that ACL2(p) provides. The benefits ACL2(p) currently provides are *faster execution* and *early feedback*, but we do not limit ACL2(p) to providing only these benefits in the future (as ACL2(p) matures, we hope that developers will find additional ways for ACL2(p) users to benefit from parallel execution; see Section 10.2 for an introduction to one possible idea). Categorization schemes similar to ours can help others that are considering parallelizing an interactive system determine the usefulness of such an effort. This chapter contains a warmup example of a proof that would benefit from parallel execution and then delves into the aforementioned categorization scheme.

7.1 Warmup Proof Example

Before going into deeper discussion of each proof attempt category, we show a simple example designed to introduce the benefits of faster execution and early feedback. Figure 7.1 shows the dependency graph for our introductory example. This dependency graph shows the relationship between each

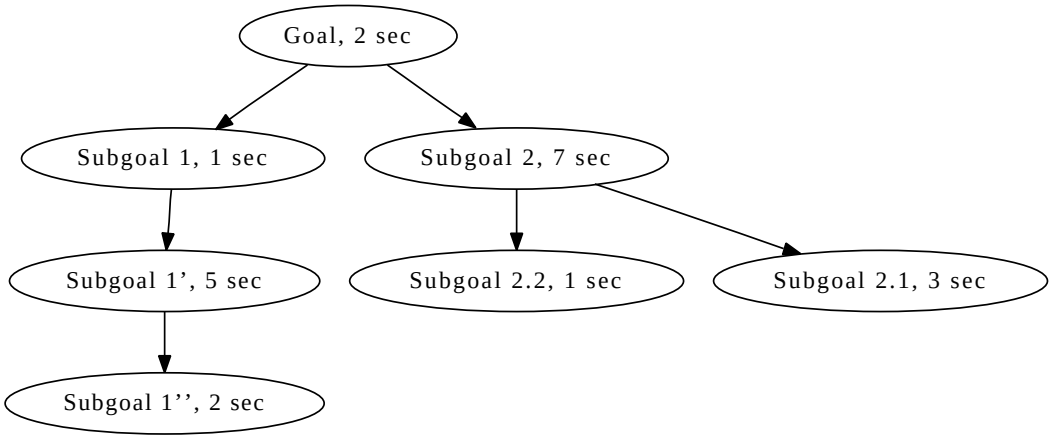

Figure 7.1: Example Introductory Subgoal Graph (times shown in seconds)

subgoal and how long each subgoal takes to complete. When executing serially, this proof attempt takes 21 seconds. As shown in the graph, two seconds into the proof, *Goal* splits into *Subgoal 1* and *Subgoal 2*. If we were to execute the proofs of *Subgoal 1* and *Subgoal 2* in parallel, we could reduce the time required to complete the whole proof attempt. Indeed, if we parallelize computation each time the proof splits (i.e., also when *Subgoal 2* splits into *Subgoal 2.1* and *Subgoal 2.2*), the total execution time could decrease (on a machine with three or more CPU cores) from 21 seconds to the time it takes to complete the proof's critical path, 12 seconds (the critical path of the proof in this example involves completing the executions of *Goal*, *Subgoal 2*, and *Subgoal 2.1*).

The second benefit of parallel execution involves providing feedback to the user sooner than could occur if the proof were to execute serially. In

the serial version of ACL2, any feedback obtained from attempting to prove *Subgoal 1* will not occur until after the proof of *Subgoal 2* and its children finish (note that ACL2 proves subgoals in reverse numeric order). Under parallel execution, the proof of *Subgoal 1* could start immediately after *Goal* splits, so any feedback that *Subgoal 1* produces could be available as early as 2 seconds into the proof attempt. This would be significantly sooner than what happens under serial execution, when feedback for *Subgoal 1* and its children would, at earliest, be available 13 seconds into the proof attempt. Provided we used more than one thread to parallelize the execution of the waterfall, this type of early feedback could occur even on a machine with only a single CPU core.

7.2 Categorization of Proofs Based on Benefits from Parallel Execution

The remainder of this chapter contains examples, taken from the ACL2 regression suite, and performance results, for each of the following proof attempt categories:

Category I Those that are so short-lived that parallel execution is useless,

Category II Those that require a non-trivial amount of time and contain a mostly linear tree of proof dependencies where independent subgoals that can be proved in parallel (typically derived from reasoning about functions containing conditionals) do not occur early enough in the proof, preventing the execution time from improving and also preventing early

feedback,

Category III Those that require a non-trivial amount of time and contain a mostly linear tree of proof dependencies, but with quick and independent subgoals that can be proved in parallel occurring early enough such that the proof attempt may still provide early feedback (despite relatively little change in execution time), and

Category IV Those that require a non-trivial amount of time and have enough time-consuming independent proof subgoals such that the execution time of the proof can benefit and early feedback can be provided.

Note that if parallelizing the execution of an ACL2 proof attempt can provide faster execution, then it will always be able to provide early feedback.

7.2.1 Category I: Short-lived Proofs

Proof attempts that take little time to finish will benefit very little from parallel execution. Some examples of such proofs are the associativity of append (shown in Figure 7.2) and the identity of the double reverse of a list (shown in Figure 7.3).

One possible motivation for using a waterfall parallelism mode is that, by using such a mode, users receive a subset of the output, known as ACL2(p) *key checkpoints* (see Section 3.3 for a discussion of this output). These key checkpoints are similar to the key checkpoints presented by the pre-existing

ACL2 feature *gag mode* (described in the ACL2 Manual [1] in documentation topic "gag-mode"). So, if users just want a similar subset of the output for these short-lived proofs, they can use gag mode, without using parallel execution.

```
(defthm assoc-of-app
        (equal (append (app a b) c)
               (append a (app b c))))
```

Figure 7.2: Theorem *Associativity of Append*

```
(defthm rev-rev
        (implies (true-listp x)
                 (equal (rev (rev x)) x)))
```

Figure 7.3: Theorem *Identity of Double-reversing a List*

7.2.2 Category II: Mostly Linear Proofs with Late Case-splitting

Some proofs are inherently sequential for most of their execution and have a small opportunity for parallelism right before they finish. In this section, we show an example of a proof that contains case-splits (which occur when reasoning about code that uses conditionals), but the case-splits occur so late in the proof, that any speedup that could be gained from proving these subgoals in parallel is negligible compared to the overall proof time. Additionally, since parallel proofs of these subgoals can not begin until near the end of the proof attempt, none of the checkpoints that could be proved in parallel would be displayed to users significantly sooner than if they did not use paral-

lel execution. As such, this class of proof attempts also does not benefit from our implementation of parallel execution.

Case Study: Theorem Ste-thm-weaken-strengthen As an example of this type of proof, we examine theorem *ste-thm-weaken-strengthen* from book *workshops/1999/ste/inference.lisp* (which, like all of the books mentioned in this dissertation, is part of the library of distributed ACL2 books). When executing in a single thread, this theorem requires 65.78 seconds, but it experiences almost no speedup when executing in parallel (a speedup of approximately 1.06 is observed with `resource-based` waterfall parallelism). We can conclude from figures 7.4 and 7.5 that almost all of the time is spent on *Goal'6'*, and once that goal is broken into subgoals (specifically *Subgoal 1* through *Subgoal 8*), the proof finishes almost immediately. Since *Goal'6'* is just a refinement of the original conjecture and not a subgoal resulting from a case-split upon that conjecture, no checkpoints could be presented earlier because of parallel execution until after *Goal'6'* generates its subgoals (as occurred 64.9 seconds into the log shown in Figure 7.5). Therefore, this proof attempt will not meaningfully benefit from our implementation of parallel execution.

7.2.3 Category III: Mostly Linear Proofs with Early Case-splitting

In this section, we discuss and present proof attempts that are mostly sequential in nature but can still benefit from parallel execution. A proof will typically case-split multiple times, and if the splits occur early enough with

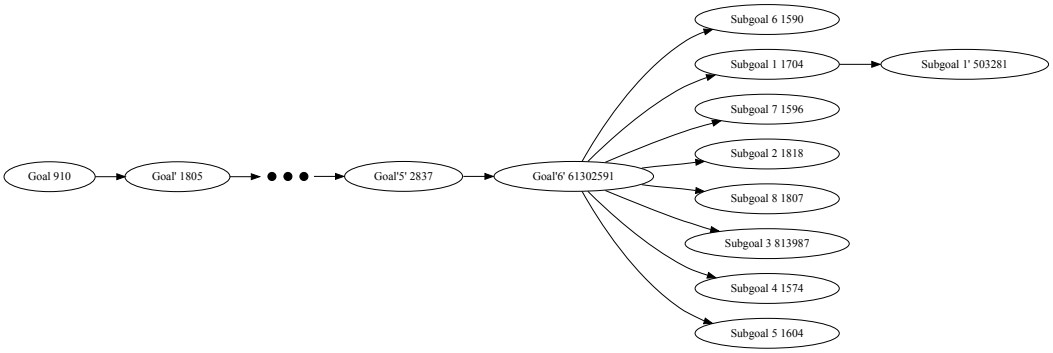

Figure 7.4: Proof Dependency Tree for Theorem *Ste-thm-weaken-strengthen* (times shown in microseconds)

waterfall parallelism enabled, users can benefit from the immediate printing of checkpoints that occurs once a checkpoint is discovered (see Section 3.3 for an introduction to checkpoints). So, there may not be much improvement in performance. Despite this, if the proof attempt of a second subgoal yields a checkpoint, and an earlier subgoal proof takes a while, users will see the second subgoal's checkpoint much sooner than if waterfall parallelism had been disabled.

Case Study: Theorem R-lte-r-deftraj-r-lte-r-deftrajs A proof that showcases the concepts of *Category III* is theorem *r-lte-r-deftraj-r-lte-r-deftrajs*, found in book *workshops/1999/ste/inference.lisp* (this theorem is a near miss in terms of being an exact example, because serial ACL2 processes *Subgoal *1/2* before it processes *Subgoal *1/1*; however, this theorem still demonstrates the right ideas). Figure 7.7 shows the proof log with timing information. Note that the proof very quickly begins *Subgoal *1/1'*, computes for about 24 sec-

```
At time 0.003951 sec, starting: Goal
At time 0.005163 sec, starting: Goal'
At time 0.007534 sec, starting: Goal''
At time 0.014035 sec, starting: Goal'''
At time 0.016123 sec, starting: Goal'4'
At time 0.019646 sec, starting: Goal'5'
At time 0.023569 sec, starting: Goal'6'
At time 64.912506 sec, starting: Subgoal 4
At time 64.913190 sec, starting: Subgoal 3
At time 64.913470 sec, starting: Subgoal 5
At time 64.913700 sec, starting: Subgoal 1
At time 64.913840 sec, starting: Subgoal 6
At time 64.914030 sec, starting: Subgoal 8
At time 64.915450 sec, finished: Subgoal 4
At time 64.916060 sec, starting: Subgoal 1'
At time 64.916400 sec, finished: Subgoal 8
At time 64.916490 sec, finished: Subgoal 5
At time 64.916900 sec, finished: Subgoal 6
At time 64.928350 sec, starting: Subgoal 2
At time 64.930520 sec, finished: Subgoal 2
At time 64.930940 sec, starting: Subgoal 7
At time 64.933180 sec, finished: Subgoal 7
At time 65.430910 sec, finished: Subgoal 1'
At time 65.431030 sec, finished: Subgoal 1
At time 65.783530 sec, finished: Subgoal 3
At time 65.784256 sec, finished: Goal'6'
At time 65.784400 sec, finished: Goal'5'
At time 65.784450 sec, finished: Goal'4'
At time 65.784485 sec, finished: Goal'''
At time 65.784530 sec, finished: Goal''
At time 65.784560 sec, finished: Goal'
At time 65.784610 sec, finished: Goal
```

Figure 7.5: Subgoal Timing Information for Theorem *Ste-thm-weaken-strengthen*

onds and then splits into fifteen cases. With waterfall parallelism enabled, if *Subgoal *1/2* or *Subgoal *1/2'* had failed to prove and generated a checkpoint, that checkpoint would have been immediately available to the user, instead of requiring the user to wait 24 seconds to receive the feedback. We also show the graphical version of the subgoal graph for this theorem in Figure 7.6.

Figure 7.8 shows the percentage of time that the CPU cores spend idle (as measured by examining the contents of Linux file */proc/stat*) during the proof of theorem *r-lte-r-deftraj-r-lte-r-deftrajs*, on *older-8-core-nht* (see Section 5.2.1 for this machine's specifications). Since the proof contains a critical path that dominates the proof's execution time, the CPU cores are idle most of the time. However, since there is a brief moment about 21 seconds into the proof's execution where the proof generates many subgoals to prove in parallel, the idle CPU core percentage drops 21 seconds after the proof starts.

The reader may observe that the total time taken for the proof in Figure 7.7 is about 32 seconds, but that the total time reported in Figure 7.8 is about 36 seconds. This is because we completely disabled garbage collection, which is single-threaded, for the figures that show the percentage of time that CPU cores spend idle. The reasons for this are discussed further in the next section, under the heading "Case Study: JVM Theorem *[2b]*."

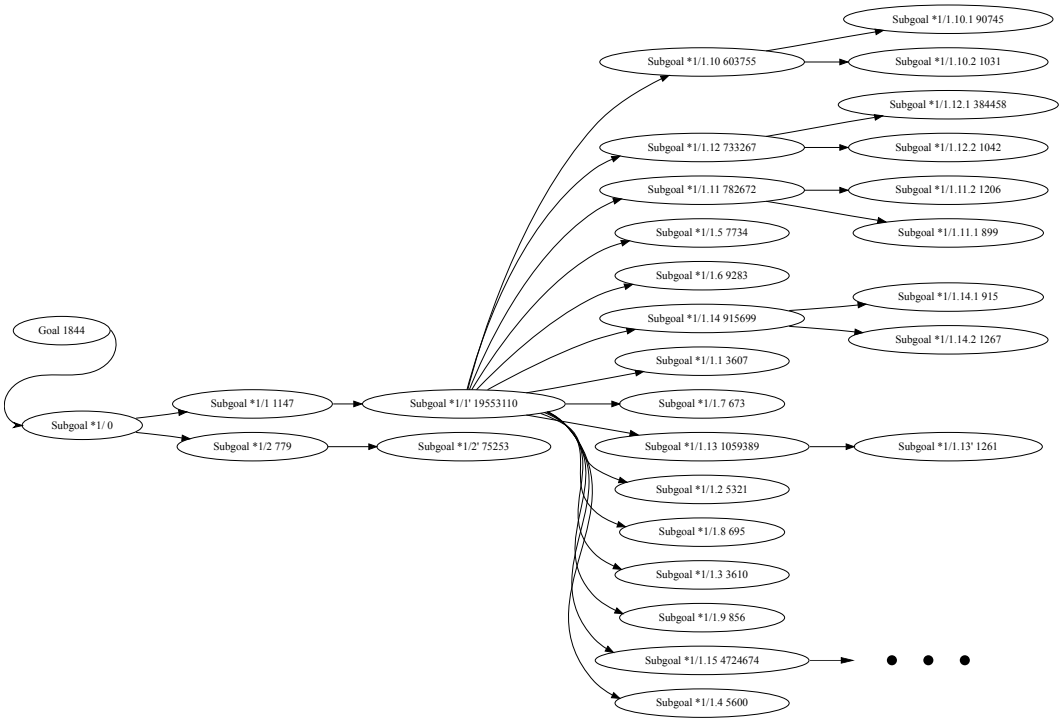

Figure 7.6: Proof Dependency Tree for Theorem *R-lte-r-deftraj-r-lte-r-deftrajs* (times shown in microseconds)

```
At time 0.018186 sec, starting: Goal
At time 0.021499 sec, finished: Goal
At time 0.027786 sec, starting: Subgoal *1/2
At time 0.028351 sec, starting: Subgoal *1/1
At time 0.028825 sec, starting: Subgoal *1/2'
At time 0.030335 sec, starting: Subgoal *1/1'
At time 0.122655 sec, finished: Subgoal *1/2'
At time 0.122816 sec, finished: Subgoal *1/2
At time 23.644804 sec, starting: Subgoal *1/1.11
At time 23.648764 sec, starting: Subgoal *1/1.14
At time 23.649048 sec, starting: Subgoal *1/1.13
At time 23.649273 sec, starting: Subgoal *1/1.8
At time 23.649504 sec, starting: Subgoal *1/1.10
At time 23.649689 sec, starting: Subgoal *1/1.9
At time 23.650867 sec, finished: Subgoal *1/1.8
At time 23.650984 sec, starting: Subgoal *1/1.15
At time 23.651323 sec, starting: Subgoal *1/1.12
At time 23.654343 sec, starting: Subgoal *1/1.7
At time 23.654753 sec, finished: Subgoal *1/1.9
⋮
At time 28.752512 sec, starting: Subgoal *1/1.15
At time 28.755407 sec, starting: Subgoal *1/1.15'
At time 33.420930 sec, starting: Subgoal *1/1.15''
At time 33.429035 sec, starting: Subgoal *1/1.15'''
At time 33.431170 sec, starting: Subgoal *1/1.15'4'
At time 36.304447 sec, finished: Subgoal *1/1.15'4'
At time 36.304626 sec, finished: Subgoal *1/1.15'''
At time 36.304714 sec, finished: Subgoal *1/1.15''
At time 36.304760 sec, finished: Subgoal *1/1.15'
At time 36.304806 sec, finished: Subgoal *1/1.15
At time 36.304890 sec, finished: Subgoal *1/1.15
At time 36.304962 sec, finished: Subgoal *1/1'
At time 36.305008 sec, finished: Subgoal *1/1
```

Figure 7.7: Timing Information for Theorem *R-lte-r-deftraj-r-lte-r-deftrajs*

7.2.4 Category IV: Proofs with Time-Consuming and Independent Subgoals

Many time-consuming theorems exhibit heavy use of case-splitting during their proofs. When parallelizing the execution of a theorem prover, an implementor should target these very proofs. Some examples of such proofs

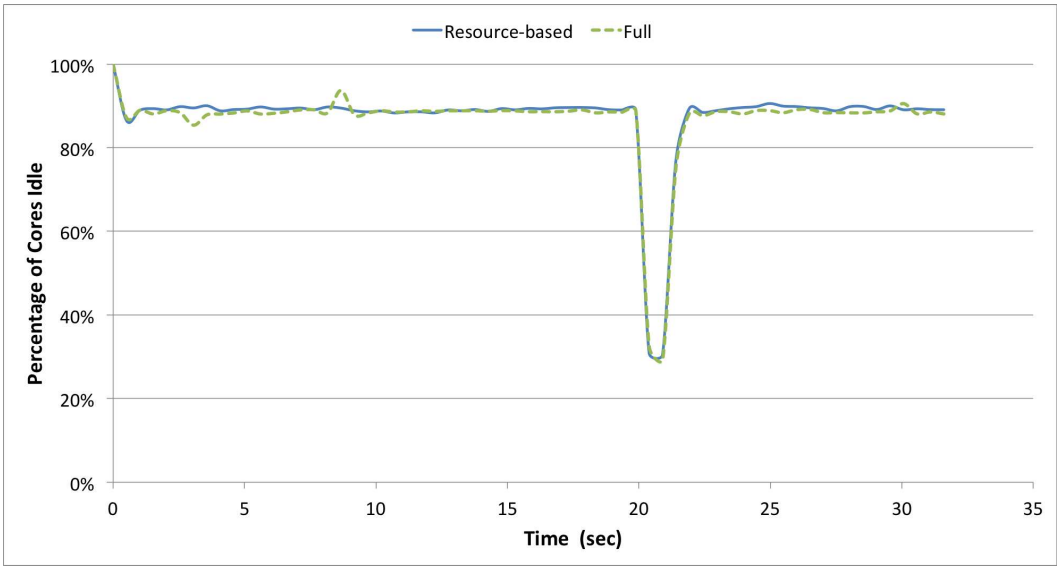

Figure 7.8: Percentage of Time Spent Idle for Theorem *R-lte-r-deftraj-r-lte-r-deftrajs*

can be found in a book containing proofs about the Java Virtual Machine, book *models/jvm/m5/apprentice.lisp* [38]. One of the best examples of this category is theorem *[2b]*, which could potentially speedup by a factor of 209x on a machine containing an arbitrarily large number of available CPU cores. In this section, we show (1) a pair of examples that experience close to linear amounts of speedup with respect to the number of CPU cores on all of our test machines, (2) further details of proof *[2b]*, which experiences a 6.65x speedup on an eight core machine and a speedup of 13.77x on a twenty core machine, (3) a proof that would perform better if we knew the critical path of the proof ahead of time and prioritized that path, and (4) a proof that lead us to one way that we improved `resource-based` waterfall parallelism.

```
(defthm ideal-8-way
  (and (f1 x) (f2 x) (f3 x) (f4 x) (f5 x) (f6 x) (f7 x) (f8 x))
  :otf-flg t)
```

Figure 7.9: Theorem *Ideal-8-way*

Case Study: Theorem Ideal-8-way and Theorem Ideal-40-way The first theorems we present for this category, *ideal-8-way* (shown in Figure 7.9) and *ideal-40-way* (shown in Figure 7.10), are designed to scale linearly with respect to the number of CPU cores in the system. The proofs simply involve calling functions (named *f1* through *f40*) that count down from a very large number and test that the value returned is not equal to a particular constant. Each of the subgoals provided in the proof is proved by execution, and the overhead for this proof is minimal. Through running these proofs, we learn the best speedup that we can hope to achieve with any real ACL2 proof. *Ideal-8-way* experiences a speedup of 7.94x on the eight core machine *older-8-core-nht* and a speedup of 7.96x on the twenty core machine *modern-20-core-2ht* (see Section 5.2.3 for this machine's specifications). *Ideal-40-way* experiences a speedup of 7.57x on *older-8-core-nht* and 18.93x on *modern-20-core-2ht*. These speedups are close to the amount of speedup that would occur in programs that scale linearly with the number of CPU cores in the system. Thus, ACL2(p) is performing well on our example theorems designed for parallel execution.

We include in Figure 7.11 the output from proving *ideal-8-way* in parallel on *older-8-core-nht* (the *ideal-40-way* output is very similar, just longer). We also include a graph of the percentage of time that CPU cores spent idle for theorem *ideal-40-way* on *older-8-core-nht* in Figure 7.12.

```
(defthm ideal-40-way
  (and (f1 x) (f2 x) (f3 x) (f4 x) (f5 x) (f6 x) (f7 x) (f8 x)
       (f9 x) (f10 x) (f11 x) (f12 x) (f13 x) (f14 x) (f15 x)
       (f16 x) (f17 x) (f18 x) (f19 x) (f20 x) (f21 x) (f22 x)
       (f23 x) (f24 x) (f25 x) (f26 x) (f27 x) (f28 x) (f29 x)
       (f30 x) (f31 x) (f32 x) (f33 x) (f34 x) (f35 x) (f36 x)
       (f37 x) (f38 x) (f39 x) (f40 x))
  :otf-flg t)
```

Figure 7.10: Theorem *Ideal-40-way*

```
At time 0.000161 sec, starting: Goal
At time 0.000711 sec, starting: Subgoal 8
At time 0.000825 sec, starting: Subgoal 7
At time 0.001032 sec, starting: Subgoal 5
At time 0.001288 sec, starting: Subgoal 2
At time 0.001541 sec, starting: Subgoal 1
At time 0.010421 sec, starting: Subgoal 6
At time 0.011706 sec, starting: Subgoal 3
At time 0.019221 sec, starting: Subgoal 4
At time 18.704100 sec, finished: Subgoal 1
At time 18.704758 sec, finished: Subgoal 8
At time 18.712820 sec, finished: Subgoal 5
At time 18.721780 sec, finished: Subgoal 3
At time 18.732887 sec, finished: Subgoal 2
At time 18.746054 sec, finished: Subgoal 4
At time 18.749727 sec, finished: Subgoal 6
At time 18.767088 sec, finished: Subgoal 7
At time 18.767320 sec, finished: Goal
```

Figure 7.11: Timing Information for Proving Theorem *Ideal-8-way* in Parallel

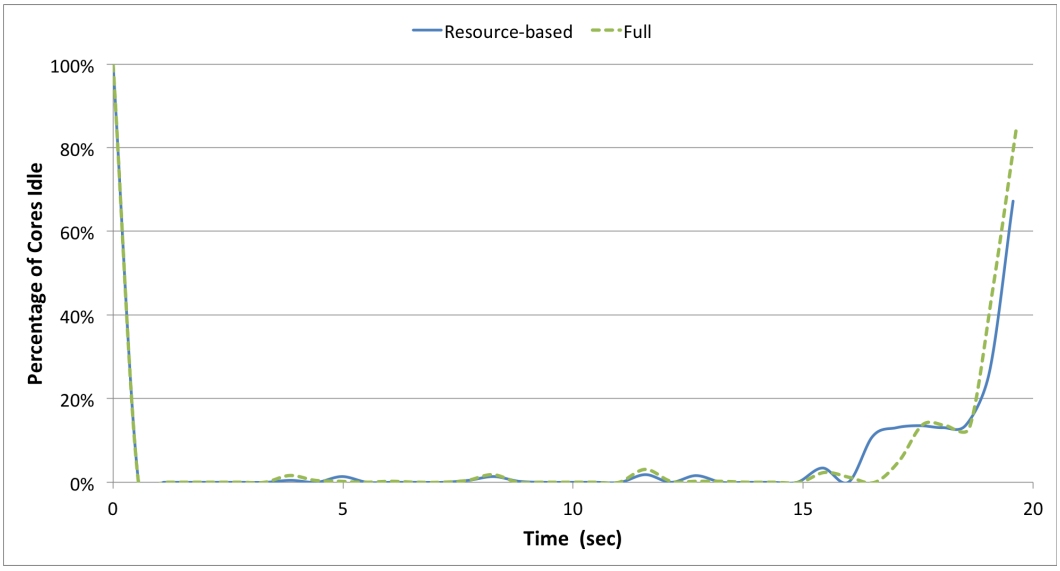

Figure 7.12: Percentage of Time Spent Idle for Theorem *Ideal-40-way*

Case Study: JVM Theorem [2b] Many of the proofs in this category have lots of independent subgoals that can be processed in parallel. Theorem *[2b]*, which has 2,858 subgoals, is a clear example of this. We include the dependency graph for *[2b]* in Figure 7.13. Along the left side of this figure, we show the broad set of subgoals ripe for parallel execution. We magnify a portion of the subgoals below *Subgoal 7* so that the reader can see the further parallelism available underneath the top-level. This figure should also give a feel for the amount of breadth, as opposed to depth, that can occur in proofs with lots of subgoals.

While we run most of the tests in this dissertation with the default ACL2(p) garbage collection configuration, the figures that show the percentage

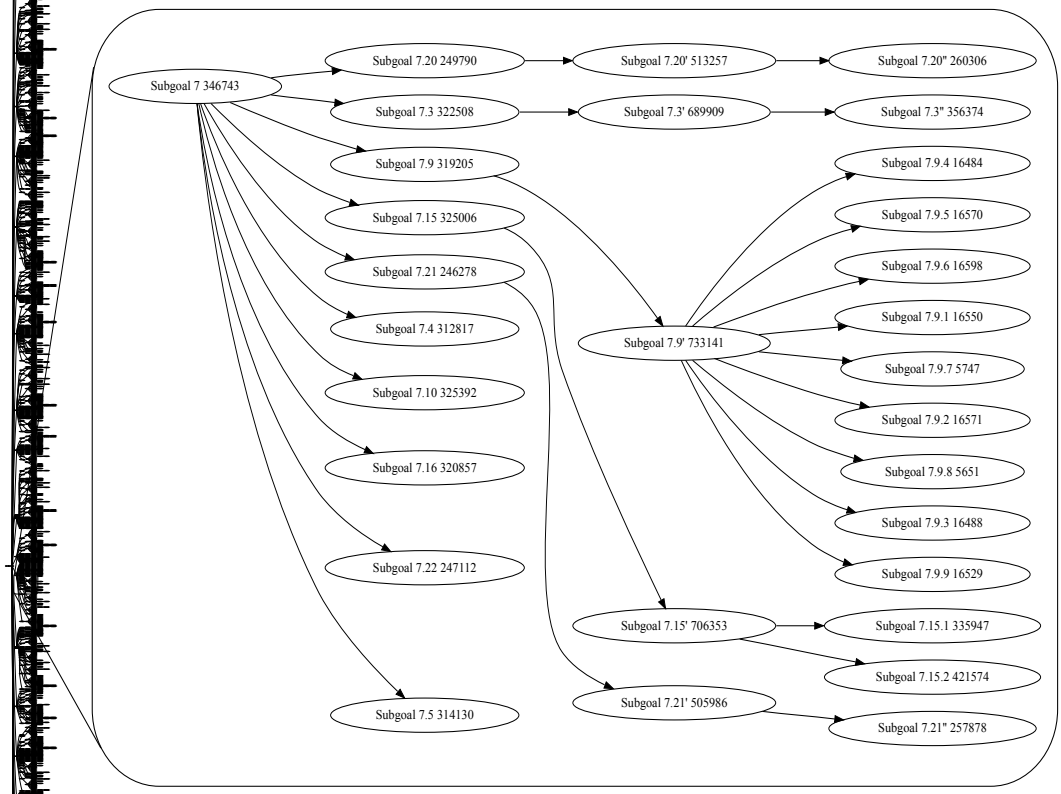

Figure 7.13: Proof Dependency Tree for JVM Theorem *[2b]* (times shown in microseconds)

of time that CPU cores spend idle have garbage collection entirely disabled (implemented by setting the garbage collection threshold to 24 gigabytes). We now explain why. Garbage collection in Lisp and ACL2 requires suspending all user-level threads and running the garbage collector in a single-thread. As a result, the charts that show the time that CPU cores spend idle will show a spike in idle time whenever the garbage collector runs. Figure 7.14 shows the percentage of time that CPU cores spend idle when proving theorem *[2b]* with the garbage collector enabled. Notice that the garbage collector runs approximately every 10 seconds. Figure 7.15 shows the performance results for the same theorem, but with garbage collection disabled. This second graph presents a much cleaner picture of how busy the CPU cores are. This cleaner picture becomes more important when examining more surprising results, such as those found in Figure 7.20.

Before leaving theorem *[2b]*, one might wonder whether removing the garbage collector from its proof causes the proof to achieve speedup closer to a factor of 8x (instead of 6.65x). Indeed, with garbage collection disabled, the speedup increases to 7.48x. While it would therefore be optimal for the performance of a single ACL2(p) process to run with an even higher garbage collection threshold, such a high threshold could cause the machine to run out of memory (the issue is further exacerbated when considering the concurrent execution of multiple ACL2(p) sessions). As such, we leave the default garbage collection threshold at two gigabytes for now and leave development of implementations that further increase the garbage collection threshold as

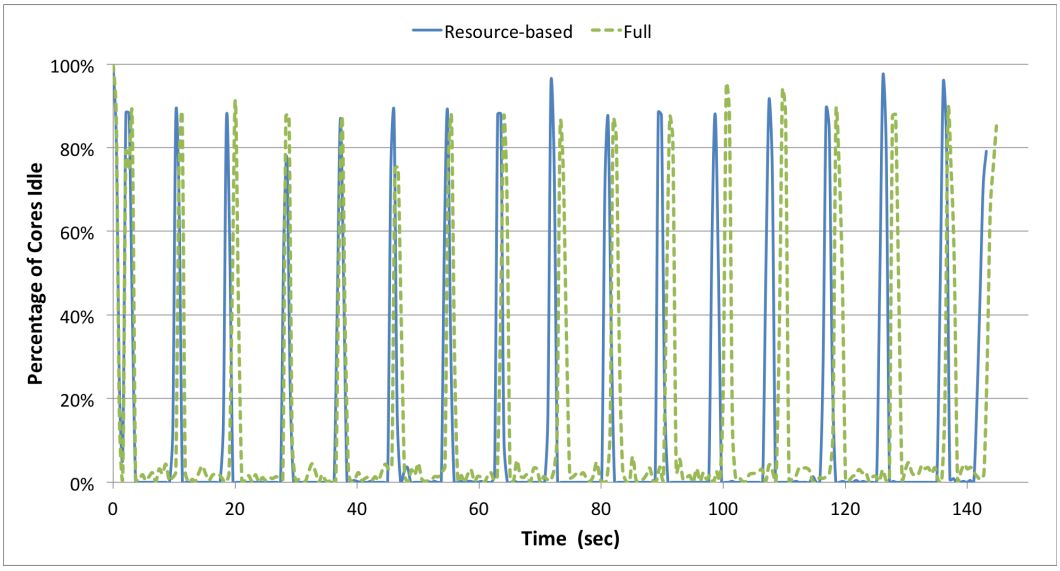

Figure 7.14: Percentage of Time Spent Idle for Theorem *[2b]* with Garbage Collection Enabled

future work.

Case Study: Theorem Step2-marks-3marked-node-either-2-or-3-or-4 Some proofs have so many subgoals that proving all of the available subgoals at once, each in their own thread, would result in suboptimal performance. One principle upon which we operate is that it is disadvantageous to have more threads running than the number of hardware threads in the system (e.g., in a system with twenty cores, each of which are two-way hyper-threaded, there are forty hardware threads, and so, generally, we do not want more than forty threads running at the same time). As such, we allow subgoals to be stored in the work queue until one of the worker threads becomes idle and can

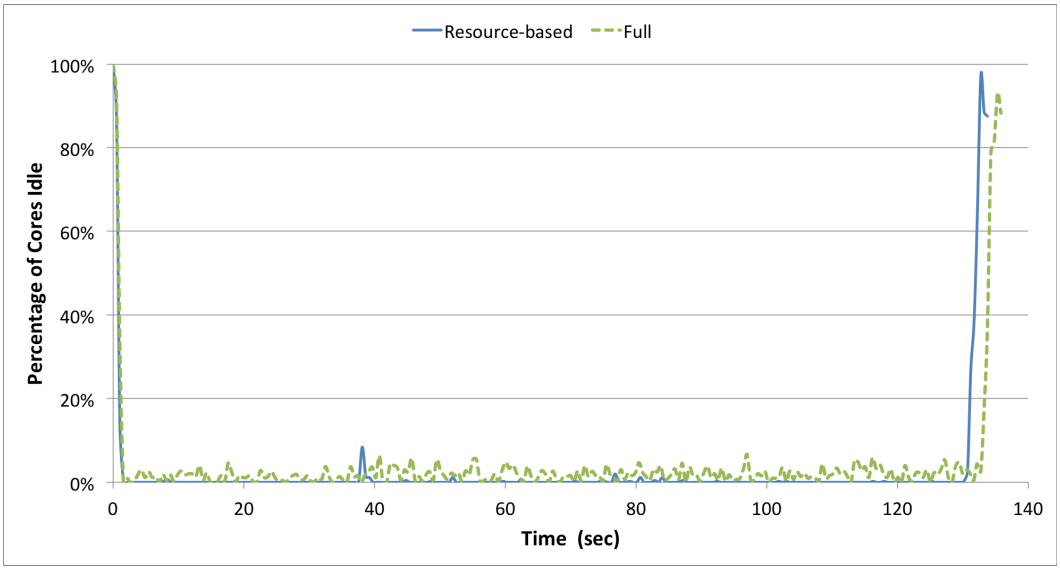

Figure 7.15: Percentage of Time Spent Idle for Theorem *[2b]* with Garbage Collection Disabled

execute the next piece of parallelism work.

In systems with a very large number of CPU cores, there is often not enough potential for parallel execution within the proof to actually have a work queue with pieces of parallelism work that aren't started immediately. However, in systems with smaller numbers of CPU cores (perhaps eight CPU cores or less), the portion of the work queue that is waiting for a worker thread and a CPU core to become available to process it is frequently non-empty. The main disadvantage of this is that *it is possible for the critical path of the proof to sit idle in the work queue.* So, as long as the critical path is sitting idly in the work queue, reaching the end of the proof attempt is delayed. We can see the effects of this by studying theorem *step2-marks-3marked-node-*

either-2-or-3-or-4, which can be found in the book *workshops/2011/verbeek-schmaltz/sources/correctness.lisp*.

We first include, in Figure 7.16, a graph that shows how many microseconds each subgoal takes and the dependencies between each subgoal. We also show the output that results from timing each subgoal and its children. The subgoal numbers appear in reverse numerical order, because ACL2 proves the subgoals in reverse order. We include, in Figure 7.17, the log that includes timing information generated with serial execution. Note that *Subgoal *1/23'*, *Subgoal *1/12'*, *Subgoal *1/9'*, and *Subgoal *1/5'* all require around ten seconds or more to complete. Also note that the colors of the log entries correspond to the colors of the nodes shown in Figure 7.16.

We also include, in Figure 7.18, the timing output for the same proof with `full` waterfall parallelism enabled. Note that *Subgoal *1/5* does not begin execution until almost ten seconds after the proof attempt begins. So even though *Subgoal *1/9* is the initial critical path, *Subgoal *1/5* becomes the problematic path and causes the proof to finish much later than it would, compared to if *Subgoal *1/5* were to begin executing immediately when the proof starts.

Figure 7.19 further confirms the idea that CPU cores are being left idle as the theorem nears the end of its proof. About halfway through the theorem's execution, the CPU cores start to run out of subgoals to process and begin to idle. This continues until the end, when only one core is being used, to prove *Subgoal *1/5'*.

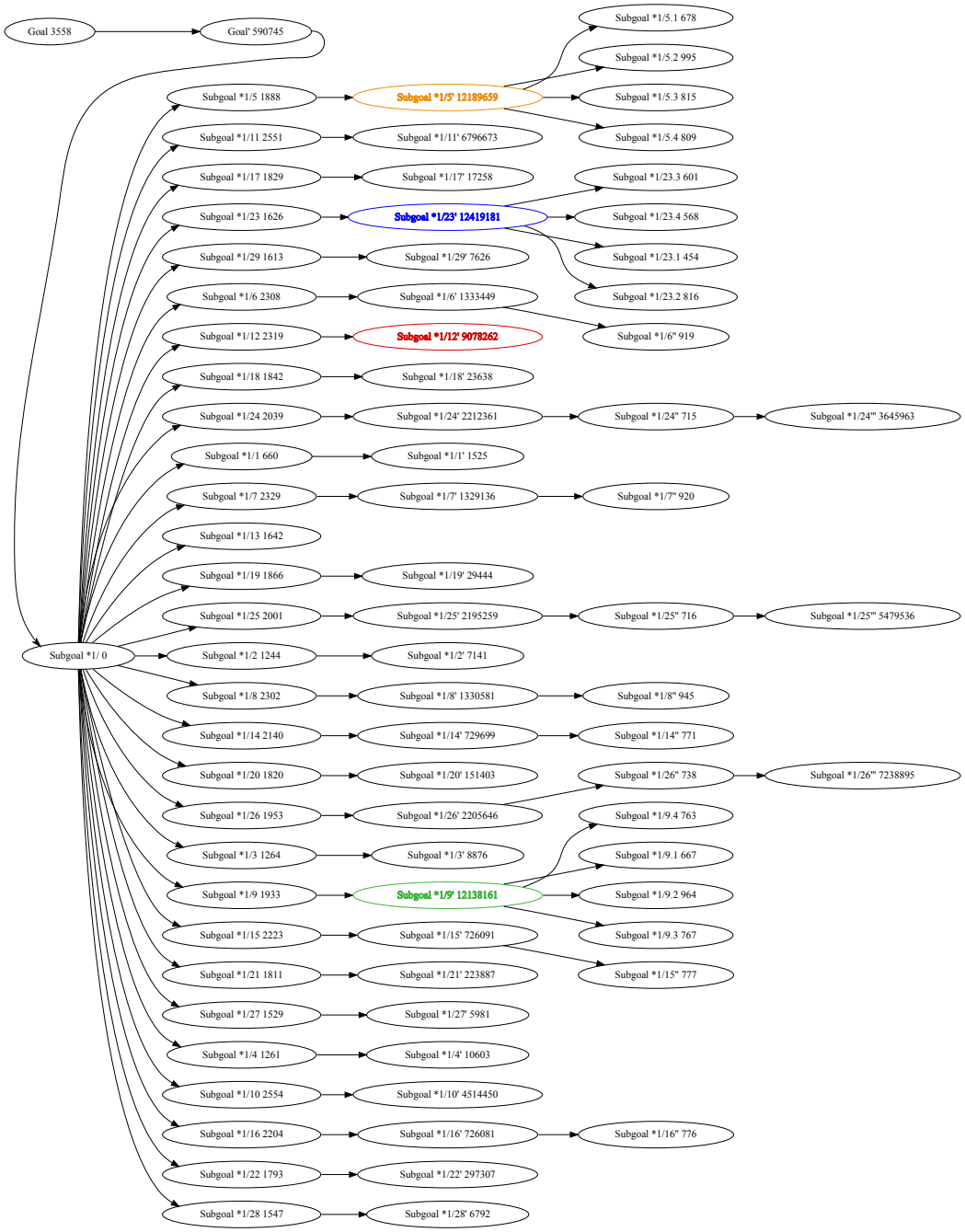

Figure 7.16: Proof Dependency Tree for Theorem *Step2-marks-3marked-node-either-2-or-3-or-4* (times shown in microseconds)

```
At time 0.012111 sec, starting: Goal
At time 0.016468 sec, starting: Goal'
At time 0.647437 sec, finished: Goal'
At time 0.647479 sec, finished: Goal
At time 0.679439 sec, starting: Subgoal *1/29
At time 0.681465 sec, starting: Subgoal *1/29'
At time 0.689781 sec, finished: Subgoal *1/29'
⋮
At time 25.302872 sec, starting: Subgoal *1/23
At time 25.304775 sec, starting: Subgoal *1/23'
⋮
At time 37.709490 sec, finished: Subgoal *1/23'
At time 37.709530 sec, finished: Subgoal *1/23
⋮
At time 40.800484 sec, starting: Subgoal *1/12
At time 40.803204 sec, starting: Subgoal *1/12'
At time 50.271072 sec, finished: Subgoal *1/12'
At time 50.271210 sec, finished: Subgoal *1/12
⋮
At time 62.102720 sec, starting: Subgoal *1/9
At time 62.105000 sec, starting: Subgoal *1/9'
⋮
At time 74.824020 sec, finished: Subgoal *1/9'
At time 74.824066 sec, finished: Subgoal *1/9
⋮
At time 79.034150 sec, starting: Subgoal *1/5
At time 79.036354 sec, starting: Subgoal *1/5'
⋮
At time 92.327320 sec, finished: Subgoal *1/5'
At time 92.327360 sec, finished: Subgoal *1/5
⋮
At time 92.364040 sec, finished: Subgoal *1/1'
At time 92.364130 sec, finished: Subgoal *1/1
```

Figure 7.17: Subgoal Timing Log for Theorem *Step2-marks-3marked-node-either-2-or-3-or-4* with `Pseudo-Parallel` Waterfall Parallelism

```
At time 0.023785 sec, starting: Goal
At time 0.028655 sec, starting: Goal'
At time 0.710664 sec, finished: Goal'
At time 0.710708 sec, finished: Goal
At time 0.744083 sec, starting: Subgoal *1/29
⋮
At time 0.746550 sec, starting: Subgoal *1/29'
⋮
At time 0.747806 sec, starting: Subgoal *1/23
⋮
At time 0.759854 sec, starting: Subgoal *1/23'
⋮
At time 1.119937 sec, starting: Subgoal *1/12
At time 1.123487 sec, starting: Subgoal *1/12'
⋮
At time 1.912167 sec, starting: Subgoal *1/9
At time 1.914921 sec, starting: Subgoal *1/9'
⋮
At time 9.781008 sec, starting: Subgoal *1/5
At time 9.783740 sec, starting: Subgoal *1/5'
⋮
At time 12.276187 sec, finished: Subgoal *1/12'
At time 12.276315 sec, finished: Subgoal *1/12
⋮
At time 14.882328 sec, finished: Subgoal *1/23'
At time 14.882380 sec, finished: Subgoal *1/23
⋮
At time 16.441063 sec, finished: Subgoal *1/9'
At time 16.441109 sec, finished: Subgoal *1/9
⋮
At time 22.921940 sec, finished: Subgoal *1/5'
At time 22.921984 sec, finished: Subgoal *1/5
```

Figure 7.18: Timing Information for Theorem *Step2-marks-3marked-node-either-2-or-3-or-4* with `Full` Waterfall Parallelism

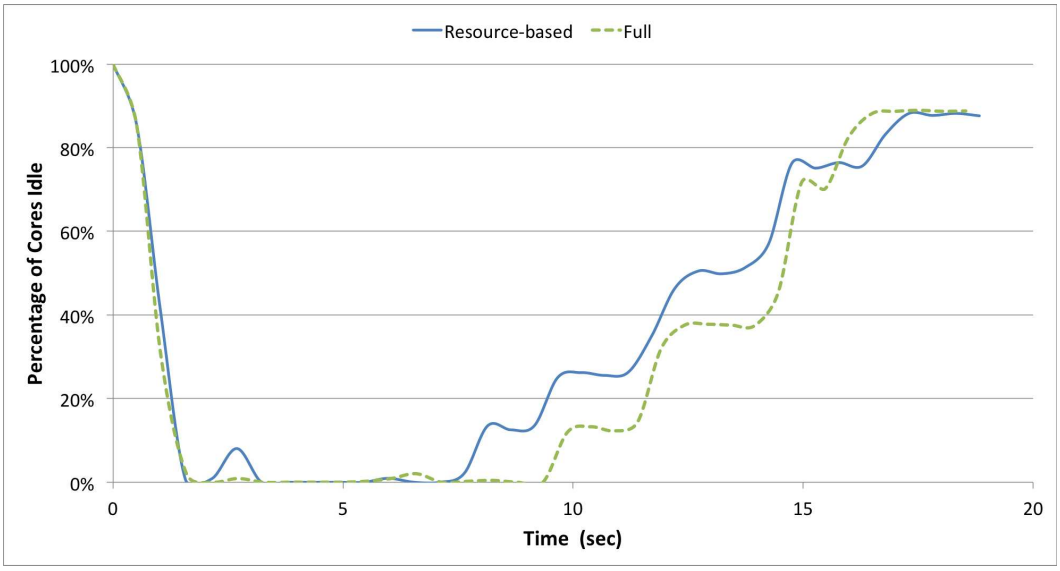

Figure 7.19: Percentage of Time Spent Idle for Theorem *Step2-marks-3marked-node-either-2-or-3-or-4*

An ideal scheduling solution would schedule the longest subgoals, *Subgoal *1/23'*, *Subgoal *1/12'*, *Subgoal *1/9'*, and *Subgoal *1/5'*, to begin executing immediately when the proof starts. However, predicting the duration that it would take to prove any particular subgoal is a known difficult problem (see Section 5.9.3 in Schumann's *Automated Theorem Proving in Software Engineering* [57]). Furthermore, even if we could determine the duration of subgoal proofs, properly prioritizing some subgoals would require a rework of the underlying futures library. We leave the development of such heuristics and modification of the underlying parallel execution system as future work.

We also created a waterfall parallelism mode that takes into account timing information from prior attempts at proving a given theorem when de-

ciding whether to parallelize execution, and perhaps this prior timing information could be used to prioritize *Subgoal *1/5*. However, for reasons discussed in Section 6.1, we leave the further refinement of this mode as future work.

Case Study: Theorem Ub-g-chain-=-g-chain-skolem-f Most theorems run fine in both `resource-based` and `full` waterfall parallelism modes. However, performance can degrade when there is a very large number of subgoals with a large dependency chain.

In earlier versions of our work, we observed that when the total amount of parallelism work allowed into the system (see sections 6.2 and 8.3 for an explanation of the notion of the "total amount of parallelism work") was 200, that theorem *ub-g-chain-=-g-chain-skolem-f* did not perform as well in `resource-based` mode as we thought possible. This theorem would hit the limit of 200, and thus, it would serialize computation for the latter part of its proof. Through experimentation, we realized that a much higher limit obtains a speedup of 5.41x, whereas the lower limit only obtains a speedup of 1.26x. As our implementation stabilizes, we continually investigate different settings for this limit and have settled, for now, upon a limit of 8,000 pieces of parallelism work. In the long term, our goal is to have as high a limit as possible, such that there is good reason to believe that it will not cause stability problems for users.

We now discuss the differences in performance between `resource-based` and `full` waterfall parallelism, using theorem *ub-g-chain-=-g-chain-skolem-f*

as a primary example. As stated in the introduction, the performance of these two modes is similar across the 200 longest theorems in the ACL2 regression suite; the `resource-based` mode achieves an average speedup of 3.69x on each theorem, and the `full` mode achieves an average speedup of 3.66x. Some theorems, like theorem *[2b]*, perform slightly better in the `resource-based` mode, while others, including *ub-g-chain-=-g-chain-skolem-f*, perform better in the `full` mode (see Table 7.1 for data that supports these claims). Figure 7.20 shows the percentage of CPU core time spent idle during a typical run of theorem *ub-g-chain-=-g-chain-skolem-f* under both modes. The line associated with `full` waterfall parallelism is about what we expect. However, the periods of non-significant idle time encountered under `resource-based` waterfall parallelism could be disconcerting. Even though other executions of this theorem typically show similar results, as Figure 7.21 shows, it is possible for the proof to be non-deterministically scheduled in a way that does not result in significant idle CPU core time during the middle of the proof. The cause for this inefficiency in `resource-based` waterfall parallelism likely lies in the limit placed upon the size of the *unassigned* part of the work queue. We postpone further discussion of this part of the work queue until Section 8.3.2, but by increasing this limit from 24 (the default value for *older-8-core-nht*) to 2000 (a number more than sufficient to allow the enqueuing of all subgoals of this proof for parallel execution), we obtain the results shown in Figure 7.22. From these results, we determine that increasing the limit upon the size of the *unassigned* section would likely benefit the performance of this proof with

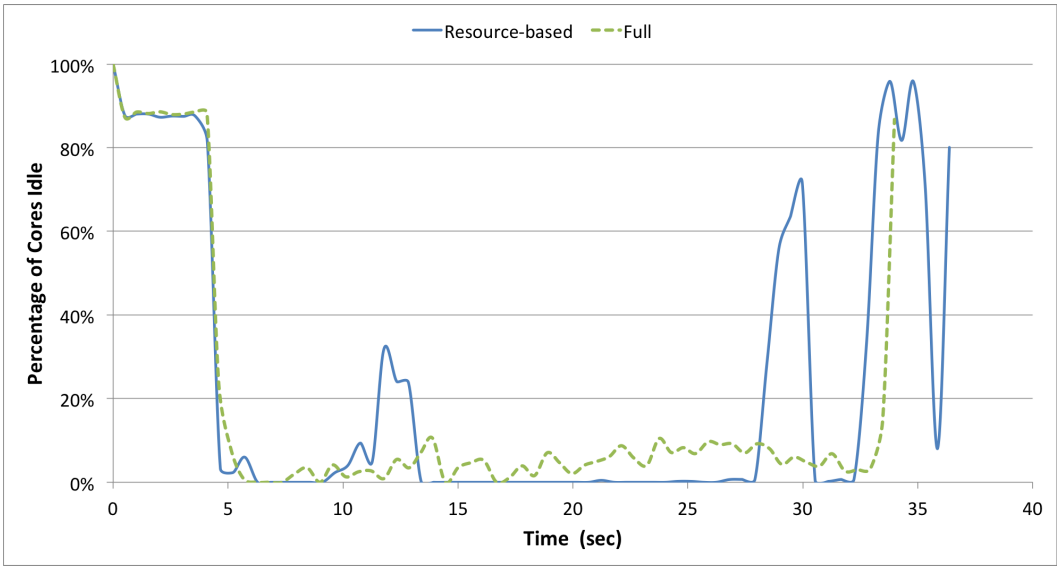

Figure 7.20: Typical Percentage of Time Spent Idle for Theorem *Ub-g-chain-=-g-chain-skolem-f*

resource-based waterfall parallelism. We leave the further tuning of this limit as future work.

7.3 ACL2(p) Proof Results

We are not interested in speedup for proof attempts that take a small amount of time (those proofs that fall into *Category I*, which is explained in Section 7.2.1). However, we have obtained non-trivial speedup for many substantial proofs. In this section we present the execution time of proofs in different waterfall parallelism modes for three types of machines: (1) an older eight core machine with no hyper-threading, (2) a modern four core machine with two-way hyper-threading (for a total of eight hardware threads), and (3)

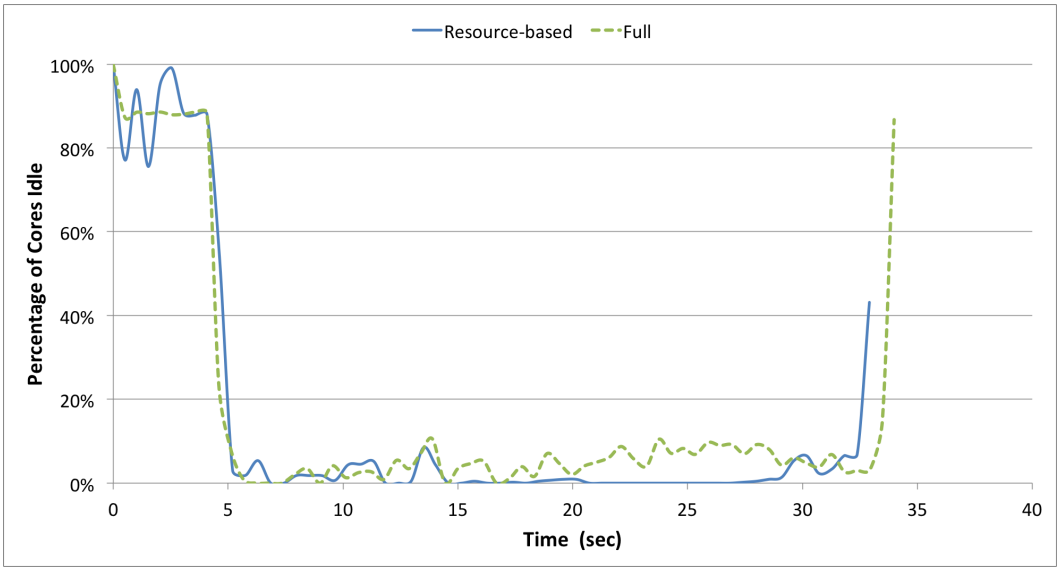

Figure 7.21: Percentage of Time Spent Idle for Theorem *Ub-g-chain-=-g-chain-skolem-f* with an Optimal Execution

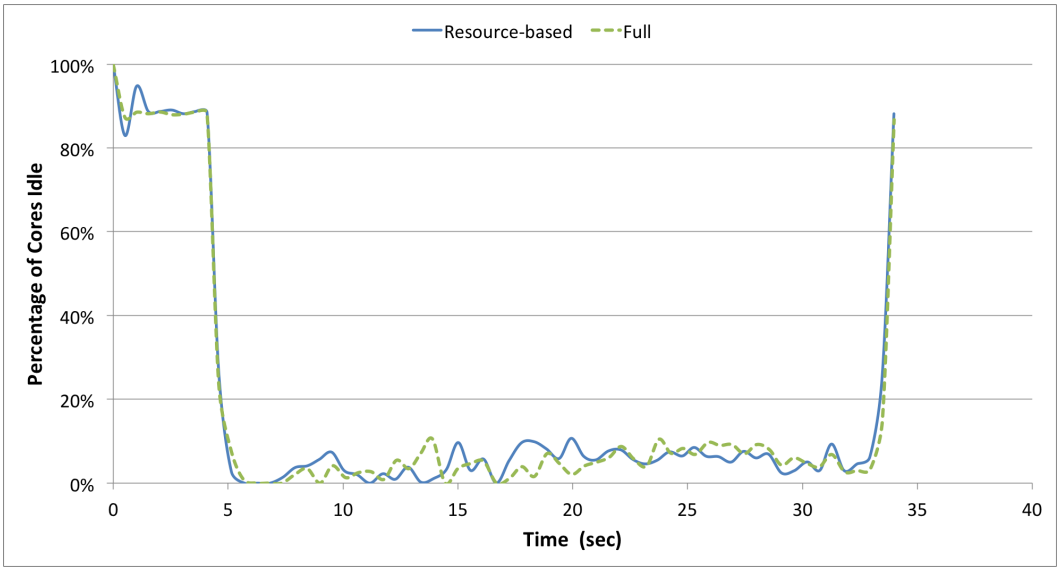

Figure 7.22: Percentage of Time Spent Idle for Theorem *Ub-g-chain-=-g-chain-skolem-f* with an *Unassigned* Size Limit of 2000

a modern twenty core machine, with ten cores per processor, and two-way hyper-threading (for a total of forty hardware threads).

In the following charts, we calculate the potential speedup for each of the longest two-hundred theorems from the ACL2 regression suite. *We define potential speedup as the amount of speedup that a proof could experience if it were run on a machine with an arbitrarily large number of CPU cores and an implementation with an insignificant amount of parallelism overhead.* For example, if a theorem has a critical path that requires 10 seconds to finish, and the entire proof requires 21 seconds, then the potential speedup for that proof with an arbitrarily large number of CPU cores is 21/10, which is a speedup factor of 2.1x.

We then use this potential speedup to calculate each theorem's "grade". *We define the "grade" as the theorem's experimental speedup divided by the theorem's potential speedup for a particular machine.* To further understand the notion of a "grade", consider the following two examples:

- A theorem that has a potential speedup of 100x, is executing on an 8-core machine, and has an experimental speedup of 4x would receive a "grade" of 50%.
- A theorem that has a potential speedup of 2x, is executing on an 8-core machine, and has an experimental speedup of 1.8x would receive a "grade" of 90%.

In the tables that contain only the longest twenty-five theorems, we label each theorem as *Category I*, *II*, *III*, or *IV*. Of the twenty-one theorems not designed for parallel execution (i.e., excluding the theorems that begin with "ideal"), seventeen fall into *Category IV*. This implies that many lengthy proof attempts can benefit from parallel execution in both performance and opportunities for seeing checkpoints sooner. Of the remaining four proofs, only one of them falls into *Category II*, and the other three proofs fall into *Category III*. Thus, we assume that, generally speaking, if a proof requires a non-trivial amount of time to finish, that it can benefit from parallel execution.

To determine the category of a proof, one must answer three questions. First, is the proof so short that parallel execution is useless? If so, the proof attempt belongs in *Category I*. If *Category I* were relevant to our categorization of the theorems we present in this section, perhaps we would use a threshold of five seconds to determine whether a proof's execution time is non-trivial. However, one can easily make the argument that anything that requires more than one second to finish can cause distress to users in an interactive setting, and thus, be worth parallelizing. The second determination is whether a proof belongs in *Category IV*. For this, we decide that if the potential speedup was greater than a factor of four, that the proof belonged in this category. Again, the choice of the exact value for this threshold is a value-judgement, and the reader can easily prefer a different threshold. If the theorem has not been classified as *Category I* or *IV*, then we place it in either *Category II* or *III*. To determine the theorem's correct placement, we manually examined the graph

representing the dependencies between each subgoal (e.g., the subgoal graph in Figure 7.16). From this examination, if we can tell that the critical path of the proof does not perform any case-splits until very near to the end of the theorem's execution, then the theorem belongs in *Category II*. However, as is common in most proofs, if case-splits occur before the critical path reaches its most time-consuming subpath, then the theorem goes in *Category III*. By elucidating these four categories, we provide a way to assess the usefulness of parallelizing the execution of any proof or proof attempt.

One will notice that many of the proofs receive a grade in the lower ranges of 20% - 70%. Our hypothesis is that most of the performance loss is caused by the critical path being stuck in the work queue (as demonstrated by the case study of theorem *step2-marks-3marked-node-either-2-or-3-or-4*, found in Section 7.2.4). Indeed, when we examine the parallelism dashboard for *well-founded-b-c->>*, from book *concurrent-programs/bakery/stutter2.lisp*, we see that the average work queue length is 636. This observation is consistent with our hypothesis. While our hypothesis is plausible and consistent with our observations of this theorem and others, further investigation is needed to fully understand this sub-linear scaling and is left as future work. For a preliminary discussion of some of this future work, see the paragraph labeled "resource and timing based" in Section 6.1 and the second paragraph of Section 8.3.2.1.

7.3.1 Performance on an Older Eight Core Machine

As specified in Section 5.2.1, *older-8-core-nht* is an older eight-core machine missing some of the features found on the most modern processors, such as hyper-threading. As such, of the three machines for which we present performance results, it provides the most straight-forward platform for evaluating the performance improvements of proofs using ACL2(p). Table 7.1 shows the performance details for the twenty-five proofs that have the longest execution times.

The first result from Table 7.1 that we point out is the performance of theorem *ideal-8-way*. As explained in Section 7.2.4, this is a proof that immediately splits into eight time consuming subgoals. On an eight core machine we should obtain speedup close to 8x, and we obtain a speedup of 7.94x. Figure 7.23 shows the number of theorems (of the 200 theorems with the longest execution time) that fall into each "grade" range. 92 of these theorems obtain at least 90% of the potential speedup for the eight core machine *older-8-core-nht*, 39 of the theorems achieve between 80% and 90% of the potential speedup for *older-8-core-nht*, and so forth, as displayed. Figure 7.24 groups the same 200 theorems according to the amount of speedup that each achieves. While it is good that many of the theorems obtain non-trivial speedup, perhaps the most important observation is that no theorem obtains a slowdown of more than 10% (as shown in the column labeled "0.0x - 0.9x").

Perhaps it is a little disconcerting that theorem *ideal-40-way* only obtains a speedup of 7.57x when, since the number of cores (eight) is a factor of

forty, one might expect a speedup closer to 8x. This waning in speedup could be caused by many things, including the time that elapses between when a thread finishes one subgoal and the next subgoal begins its proof. Alternatively, perhaps the underlying runtime system is temporarily scheduling two threads to execute on the same CPU core. Whatever the reason, 7.5x is perhaps a more reasonable upper bound for the speedup we could hope to achieve with more realistic theorems.

7.3.2 Performance on a Four Core Machine with Two-way Hyper-threading

Figures 7.25 and 7.26 show results similar to those found in the previous section for the four core machine *modern-4-core-2ht*. Since Linux machines are typically configured with hyper-threading enabled, and because extra hardware threads are not nearly as useful as extra CPU cores, we calculate "grades" given to theorems with respect to the number of CPU cores in the system (four).

To determine the benefits (or harm) of hyper-threading in our application with our implementation of parallel execution, we run the same set of tests, both with hyper-threading enabled and hyper-threading disabled (by disabling it at the BIOS level). The average improvement in performance (over the two-hundred longest theorems) when using hyper-threading with `resource-based` waterfall parallelism is 2.41% (there is an improvement of 2.38% with `full` waterfall parallelism). Table 7.3 shows the performance change for the twenty-

Theorem	Parallelization Type	Exp SU	Pote SU	Grade	Ser Time	Par Time	Cat
measure-obligation-4	full	4.406	96.965	55%	1062.182	241.094	IV
measure-obligation-4	resource-based	5.482	96.965	69%	1062.182	193.775	IV
[2b]	full	6.566	209.144	82%	964.119	146.833	IV
[2b]	resource-based	6.654	209.144	83%	964.119	144.898	IV
dlf->not3	full	1.570	1.640	96%	531.487	338.607	II
dlf->not3	resource-based	1.597	1.640	97%	531.487	332.756	II
[3b]	full	6.592	129.145	82%	528.751	80.208	IV
[3b]	resource-based	6.742	129.145	84%	528.751	78.421	IV
spec-body	full	5.603	21.641	70%	333.844	59.581	IV
spec-body	resource-based	5.663	21.641	71%	333.844	58.951	IV
step1-puts-dest-to-neighb...	full	4.325	6.781	64%	299.188	69.170	IV
step1-puts-dest-to-neighb...	resource-based	4.359	6.781	64%	299.188	68.644	IV
step1-puts-all-neighbors-...	full	3.715	5.060	73%	245.932	66.208	IV
step1-puts-all-neighbors-...	resource-based	3.752	5.060	74%	245.932	65.541	IV
step1-puts-all-neighbors-...	full	3.729	4.717	79%	218.365	58.552	IV
step1-puts-all-neighbors-...	resource-based	3.718	4.717	79%	218.365	58.727	IV
step1-preserves-dl->not2-...	full	3.434	4.253	81%	215.790	62.842	IV
step1-preserves-dl->not2-...	resource-based	3.463	4.253	81%	215.790	62.321	IV
step1-puts-all-neighbors-...	full	3.570	4.567	78%	211.232	59.175	IV
step1-puts-all-neighbors-...	resource-based	3.557	4.567	78%	211.232	59.390	IV
[2a]	full	6.286	44.211	79%	206.174	32.798	IV
[2a]	resource-based	6.278	44.211	78%	206.174	32.842	IV
step1-preserves-invariant...	full	4.018	5.574	72%	191.710	47.708	IV
step1-preserves-invariant...	resource-based	3.965	5.574	71%	191.710	48.348	IV
ub-g-chain-=-g-chain-skol...	full	5.411	11.720	68%	190.434	35.194	IV
ub-g-chain-=-g-chain-skol...	resource-based	5.011	11.720	63%	190.434	38.000	IV
measure-obligation-1	full	4.636	13.472	58%	182.218	39.301	IV
measure-obligation-1	resource-based	4.710	13.472	59%	182.218	38.684	IV
measure-obligation-2	full	4.682	12.565	59%	175.065	37.394	IV
measure-obligation-2	resource-based	4.741	12.565	59%	175.065	36.924	IV
fw	full	1.853	1.962	94%	167.890	90.612	III
fw	resource-based	1.870	1.962	95%	167.890	89.766	III
step1-gives-0marked-node-...	full	2.941	3.597	82%	154.316	52.475	III
step1-gives-0marked-node-...	resource-based	2.909	3.597	81%	154.316	53.056	III
ideal-40-way	full	7.473	39.697	93%	149.738	20.036	IV
ideal-40-way	resource-based	7.565	39.697	95%	149.738	19.792	IV
ideal-4-way	full	3.950	3.998	99%	149.735	37.906	IV
ideal-4-way	resource-based	3.995	3.998	100%	149.735	37.481	IV
ideal-20-way	full	6.607	19.925	83%	149.716	22.659	IV
ideal-20-way	resource-based	6.559	19.925	82%	149.716	22.826	IV
ideal-8-way	full	7.940	7.985	99%	149.705	18.854	IV
ideal-8-way	resource-based	7.937	7.985	99%	149.705	18.862	IV
temp14.00	full	2.569	3.180	81%	141.353	55.026	III
temp14.00	resource-based	2.533	3.180	80%	141.353	55.796	III
tarai_terminates_helper	full	3.464	13.626	43%	136.321	39.351	IV
tarai_terminates_helper	resource-based	4.169	13.626	52%	136.321	32.695	IV
[3a]	full	6.567	28.629	82%	121.540	18.507	IV
[3a]	resource-based	6.396	28.629	80%	121.540	19.002	IV
cases-on-th	full	6.373	66.850	80%	119.114	18.691	IV
cases-on-th	resource-based	6.749	66.850	84%	119.114	17.649	IV

Table 7.1: Performance Improvement of Twenty-Five Longest Theorems on *Older-8-core-nht*

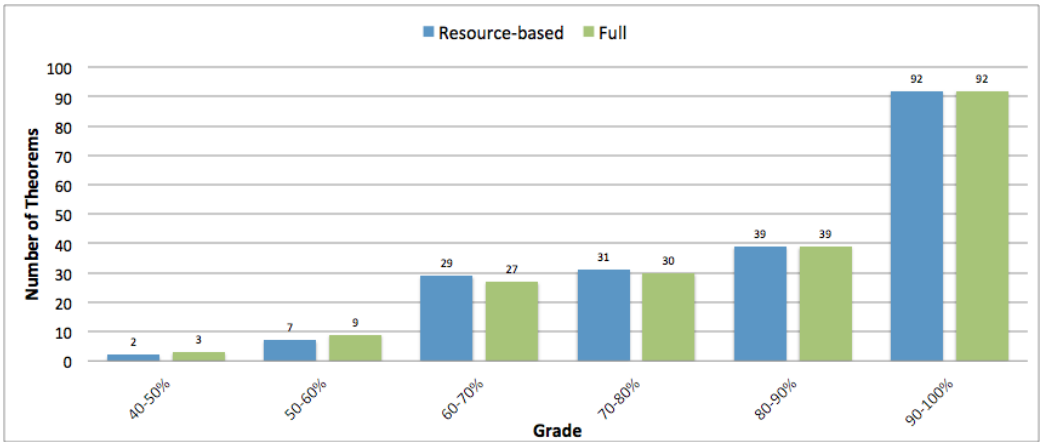

Figure 7.23: Number of Theorems (of the top 200 longest theorems) with Given Percentage of Potential Speedup on *Older-8-core-nht*

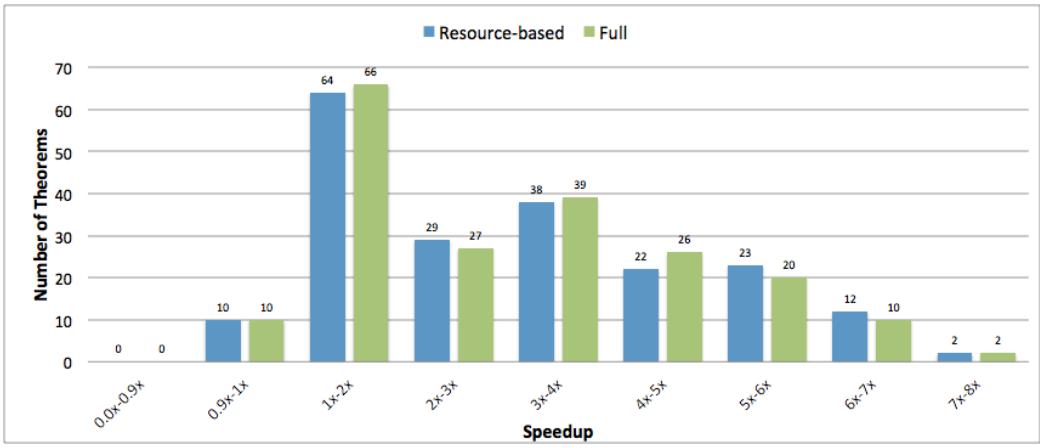

Figure 7.24: Number of Theorems (of the top 200 longest theorems) for Each Range of Experimental Speedup on *Older-8-core-nht*

five longest theorems with `full` waterfall parallelism (we could examine the statistics for `resource-based` waterfall parallelism, but the results from the `full` mode more clearly demonstrate the relevant points). Many of the theorems benefit from hyper-threading as one might expect; e.g., theorem *[2b]* runs 13.8% faster, theorem *measure-obligation-4* runs 13.2% faster, and theorem *ideal-8-way* runs 25.1% faster. However, many of the theorems run more slowly with hyper-threading enabled. A theorem that clearly shows why this is the case is theorem *ideal-4-way*, which runs 25.2% more slowly with hyper-threading enabled. Running theorem *ideal-4-way* causes four threads to immediately spawn and run, in parallel, four functions that countdown from a large number. When these functions are run with hyper-threading disabled, the operating system assigns each of them to their own core. However, when hyper-threading is enabled, the operating system likely assigns them to use different hardware threads but some of the same CPU cores. This is a sub-optimal scheduling and causes theorem *ideal-4-way* to achieve a speedup of 2.88x instead of the 3.86x speedup that it achieves with hyper-threading disabled. There are other theorems that demonstrate this issue, including theorem *fw*, which runs 15.6% more slowly with hyper-threading enabled.

7.3.3 Performance on a Twenty Core Machine with Two-way Hyper-threading

The results displayed in Table 7.4 and in figures 7.27 and 7.28 indicate that the twenty cores that *modern-20-core-2ht* provides reduce execution time more so than the eight cores that *older-8-core-nht* provides. The most surpris-

Theorem	Parallelization Type	Exp SU	Pote SU	Grade	Ser Time	Par Time	Cat
measure-obligation-4	full	3.341	96.965	84%	367.321	109.942	IV
measure-obligation-4	resource-based	3.882	96.965	97%	367.321	94.621	IV
[2b]	full	4.015	209.144	100%	352.547	87.815	IV
[2b]	resource-based	4.087	209.144	102%	352.547	86.271	IV
[3b]	full	3.999	129.145	100%	191.928	47.988	IV
[3b]	resource-based	4.081	129.145	102%	191.928	47.035	IV
dlf->not3	full	1.569	1.640	96%	181.132	115.469	II
dlf->not3	resource-based	1.389	1.640	85%	181.132	130.387	II
spec-body	full	3.761	21.641	94%	116.477	30.966	IV
spec-body	resource-based	3.791	21.641	95%	116.477	30.724	IV
step1-puts-dest-to-neighb...	full	3.286	6.781	82%	102.566	31.216	IV
step1-puts-dest-to-neighb...	resource-based	3.228	6.781	81%	102.566	31.770	IV
step1-puts-all-neighbors-...	full	2.952	5.060	74%	84.771	28.719	IV
step1-puts-all-neighbors-...	resource-based	2.968	5.060	74%	84.771	28.557	IV
ideal-4-way	full	2.884	3.998	72%	80.184	27.802	IV
ideal-4-way	resource-based	3.702	3.998	93%	80.184	21.662	IV
ideal-8-way	full	4.157	7.985	104%	80.177	19.287	IV
ideal-8-way	resource-based	4.146	7.985	104%	80.177	19.337	IV
ideal-20-way	full	3.780	19.925	94%	80.145	21.204	IV
ideal-20-way	resource-based	3.850	19.925	96%	80.145	20.815	IV
ideal-40-way	full	3.996	39.697	100%	80.131	20.053	IV
ideal-40-way	resource-based	3.941	39.697	99%	80.131	20.335	IV
step1-puts-all-neighbors-...	full	2.848	4.717	71%	75.336	26.455	IV
step1-puts-all-neighbors-...	resource-based	2.864	4.717	72%	75.336	26.305	IV
[2a]	full	3.914	44.211	98%	75.258	19.227	IV
[2a]	resource-based	3.963	44.211	99%	75.258	18.991	IV
step1-preserves-dl->not2-...	full	2.710	4.253	68%	74.156	27.361	IV
step1-preserves-dl->not2-...	resource-based	2.784	4.253	70%	74.156	26.637	IV
step1-puts-all-neighbors-...	full	2.822	4.567	71%	72.717	25.767	IV
step1-puts-all-neighbors-...	resource-based	2.837	4.567	71%	72.717	25.633	IV
ub-g-chain-=-g-chain-skol...	full	3.842	11.720	96%	65.824	17.134	IV
ub-g-chain-=-g-chain-skol...	resource-based	3.542	11.720	89%	65.824	18.585	IV
step1-preserves-invariant...	full	3.102	5.574	78%	65.610	21.152	IV
step1-preserves-invariant...	resource-based	3.026	5.574	76%	65.610	21.682	IV
measure-obligation-1	full	3.527	13.472	88%	59.360	16.829	IV
measure-obligation-1	resource-based	3.776	13.472	94%	59.360	15.719	IV
fw	full	1.608	1.962	82%	57.747	35.904	III
fw	resource-based	1.678	1.962	86%	57.747	34.419	III
measure-obligation-2	full	3.488	12.565	87%	56.496	16.196	IV
measure-obligation-2	resource-based	3.863	12.565	97%	56.496	14.624	IV
step1-gives-0marked-node-...	full	2.466	3.597	69%	53.835	21.834	III
step1-gives-0marked-node-...	resource-based	2.494	3.597	69%	53.835	21.585	III
temp14.00	full	2.259	3.180	71%	48.582	21.502	III
temp14.00	resource-based	2.295	3.180	72%	48.582	21.165	III
tarai_terminates_helper	full	2.901	13.626	73%	47.186	16.268	IV
tarai_terminates_helper	resource-based	3.250	13.626	81%	47.186	14.517	IV
[3a]	full	3.965	28.629	99%	44.342	11.182	IV
[3a]	resource-based	3.996	28.629	100%	44.342	11.096	IV
cases-on-th	full	3.975	66.850	99%	42.158	10.607	IV
cases-on-th	resource-based	4.102	66.850	103%	42.158	10.277	IV

Table 7.2: Performance Improvement of Twenty-Five Longest Theorems on *Modern-4-core-2ht*

Theorem	HT Disabled Speedup	HT Enabled Speedup	Benefit
measure-obligation-4	2.951x	3.341x	13.2%
[2b]	3.528x	4.015x	13.8%
[3b]	3.518x	3.999x	13.7%
dlf->not3	1.573x	1.569x	-0.3%
spec-body	3.314x	3.761x	13.5%
step1-puts-dest-to-neighb...	3.116x	3.286x	5.5%
step1-puts-all-neighbors-...	2.832x	2.952x	4.2%
ideal-4-way	3.856x	2.884x	-25.2%
ideal-8-way	3.322x	4.157x	25.1%
ideal-20-way	3.647x	3.78x	3.6%
ideal-40-way	3.66x	3.996x	9.2%
step1-puts-all-neighbors-...	2.869x	2.848x	-0.7%
[2a]	3.473x	3.914x	12.7%
step1-preserves-dl->not2-...	2.725x	2.71x	-0.6%
step1-puts-all-neighbors-...	2.762x	2.822x	2.2%
ub-g-chain-=-g-chain-skol...	3.27x	3.842x	17.5%
step1-preserves-invariant...	3.023x	3.102x	2.6%
measure-obligation-1	3.081x	3.527x	14.5%
fw	1.906x	1.608x	-15.6%
measure-obligation-2	3.027x	3.488x	15.2%
step1-gives-0marked-node-...	2.385x	2.466x	3.4%
temp14.00	2.233x	2.259x	1.2%
tarai_terminates_helper	2.537x	2.901x	14.3%
[3a]	3.506x	3.965x	13.1%

Table 7.3: Effects of Hyper-threading upon Twenty-Five Longest Theorems on *Modern-4-core-2ht*

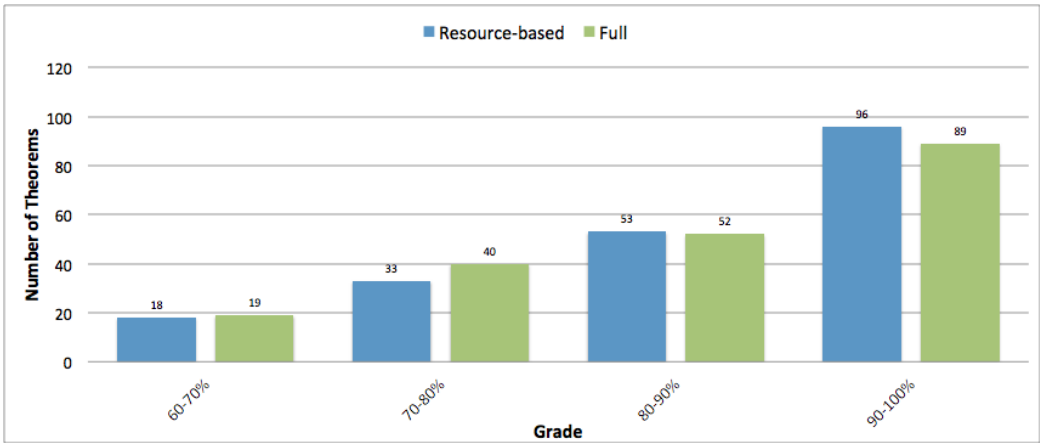

Figure 7.25: Number of Theorems (of the top 200 longest theorems) with Given Percentage of Potential Speedup on *Modern-4-core-2ht*

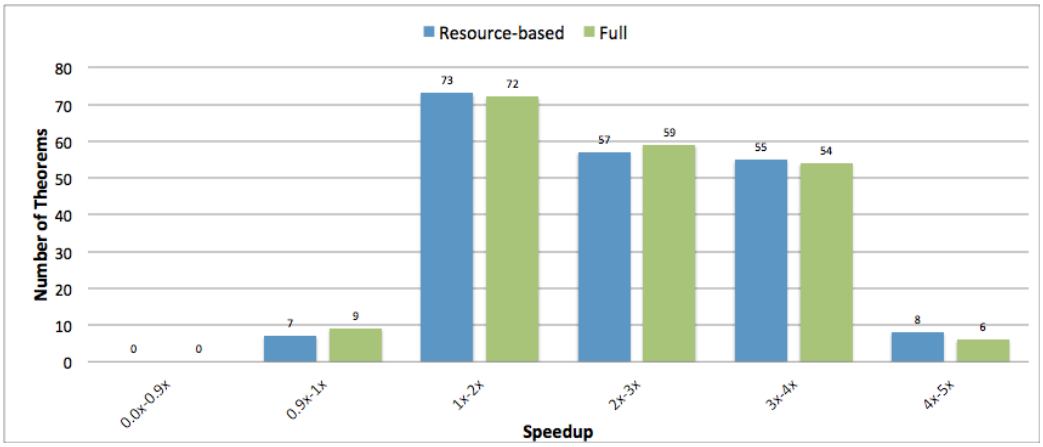

Figure 7.26: Number of Theorems (of the top 200 longest theorems) for Each Range of Experimental Speedup on *Modern-4-core-2ht*

ing result is that even though many of the proofs have a potential speedup much larger than a factor of twenty, these proofs do not obtain speedup similar to the speedup that *ideal-20-way* and *ideal-40-way* achieve. The possible causes of this lack of speedup are similar to those already presented during the introduction to Section 7.3, and further analysis concerning the lack of scalability is left as future work.

Theorem	Parallelization Type	Exp SU	Pote SU	Grade	Ser Time	Par Time	Cat
measure-obligation-4	full	6.348	96.965	32%	684.363	107.811	IV
measure-obligation-4	resource-based	7.996	96.965	40%	684.363	85.589	IV
[2b]	full	13.441	209.144	67%	654.805	48.716	IV
[2b]	resource-based	13.766	209.144	69%	654.805	47.567	IV
[3b]	full	13.306	129.145	67%	357.080	26.836	IV
[3b]	resource-based	13.973	129.145	70%	357.080	25.556	IV
dlf->not3	full	1.596	1.640	97%	348.527	218.372	II
dlf->not3	resource-based	1.591	1.640	97%	348.527	219.122	II
spec-body	full	8.246	21.641	41%	220.823	26.778	IV
spec-body	resource-based	8.445	21.641	42%	220.823	26.147	IV
step1-puts-dest-to-neighb...	full	5.531	6.781	82%	195.218	35.297	IV
step1-puts-dest-to-neighb...	resource-based	5.549	6.781	82%	195.218	35.184	IV
step1-puts-all-neighbors-...	full	4.192	5.060	83%	162.563	38.777	IV
step1-puts-all-neighbors-...	resource-based	4.201	5.060	83%	162.563	38.697	IV
step1-puts-all-neighbors-...	full	4.067	4.717	86%	144.522	35.539	IV
step1-puts-all-neighbors-...	resource-based	4.071	4.717	86%	144.522	35.503	IV
step1-preserves-dl->not2-...	full	3.333	4.253	78%	139.690	41.917	IV
step1-preserves-dl->not2-...	resource-based	3.349	4.253	79%	139.690	41.712	IV
step1-puts-all-neighbors-...	full	3.907	4.567	86%	139.582	35.730	IV
step1-puts-all-neighbors-...	resource-based	3.864	4.567	85%	139.582	36.124	IV
[2a]	full	11.759	44.211	59%	138.926	11.815	IV
[2a]	resource-based	11.521	44.211	58%	138.926	12.058	IV
ideal-4-way	full	3.952	3.998	99%	137.586	34.813	IV
ideal-4-way	resource-based	3.964	3.998	99%	137.586	34.707	IV
ideal-40-way	full	18.930	39.697	95%	137.293	7.253	IV
ideal-40-way	resource-based	18.406	39.697	92%	137.293	7.459	IV
ideal-20-way	full	14.415	19.925	72%	137.287	9.524	IV
ideal-20-way	resource-based	15.165	19.925	76%	137.287	9.053	IV
ideal-8-way	full	7.962	7.985	100%	137.284	17.242	IV
ideal-8-way	resource-based	7.945	7.985	100%	137.284	17.279	IV
ub-g-chain-=-g-chain-skol...	full	8.086	11.720	69%	127.479	15.765	IV
ub-g-chain-=-g-chain-skol...	resource-based	8.024	11.720	68%	127.479	15.888	IV
step1-preserves-invariant...	full	4.478	5.574	80%	124.736	27.857	IV
step1-preserves-invariant...	resource-based	4.574	5.574	82%	124.736	27.273	IV
measure-obligation-1	full	6.632	13.472	49%	111.362	16.793	IV
measure-obligation-1	resource-based	7.240	13.472	54%	111.362	15.382	IV
measure-obligation-2	full	6.408	12.565	51%	105.794	16.509	IV
measure-obligation-2	resource-based	6.479	12.565	52%	105.794	16.330	IV
fw	full	1.886	1.962	96%	105.082	55.715	III
fw	resource-based	1.906	1.962	97%	105.082	55.122	III
step1-gives-0marked-node-...	full	3.281	3.597	91%	102.813	31.334	III
step1-gives-0marked-node-...	resource-based	3.314	3.597	92%	102.813	31.025	III
temp14.00	full	2.843	3.180	89%	91.709	32.258	III
temp14.00	resource-based	2.855	3.180	90%	91.709	32.119	III
tarai_terminates_helper	full	4.168	13.626	31%	84.291	20.222	IV
tarai_terminates_helper	resource-based	5.596	13.626	41%	84.291	15.063	IV
[3a]	full	11.982	28.629	60%	81.884	6.834	IV
[3a]	resource-based	12.044	28.629	60%	81.884	6.799	IV
cases-on-th	full	11.708	66.850	59%	76.127	6.502	IV
cases-on-th	resource-based	11.380	66.850	57%	76.127	6.690	IV

Table 7.4: Performance Improvement of Twenty-Five Longest Theorems on *Modern-20-core-2ht*

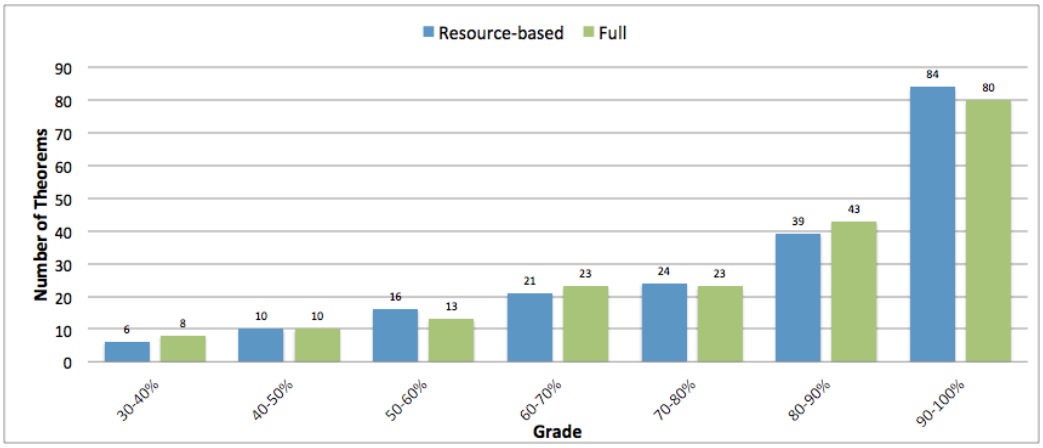

Figure 7.27: Number of Theorems (of the top 200 longest theorems) with Given Percentage of Potential Speedup on *Modern-20-core-2ht*

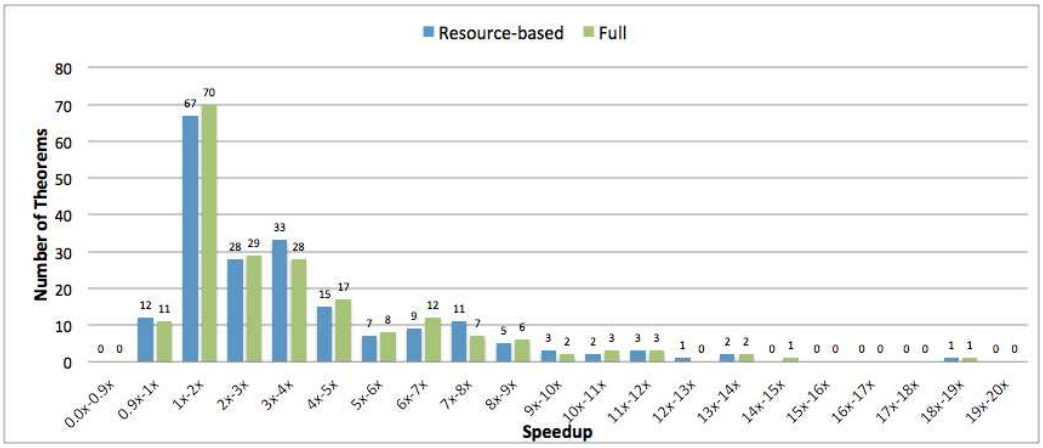

Figure 7.28: Number of Theorems (of the top 200 longest theorems) for Each Range of Experimental Speedup on *Modern-20-core-2ht*

Chapter 8

Managing the Parallelism Execution Engine

Our original parallelized ACL2 programs include simple examples such as parallelizing the execution of a doubly recursive Fibonacci function [43]. For these initial implementations, we needed a different set of heuristics and optimizations than are required for parallelizing our theorem prover. This chapter serves as documentation of some of the design decisions that we made and acts as a guide to some of the practical engineering issues that someone implementing a parallelism execution engine may encounter in their own work.

8.1 Defining a Piece of Parallelism Work

Once an ACL2 parallelism primitive is encountered and the decision to parallelize a computation is made, pieces of *parallelism work* containing the information necessary to execute the computation in parallel are added to the work queue (in the particular case of `spec-mv-let`, which uses futures, exactly one piece of parallelism work is added to the work queue). After adding the pieces of parallelism work to the work queue, the thread that encountered the ACL2 parallelism primitive signals worker threads to consume and execute the work, might continue along with some of its own work (as is the case with

`spec-mv-let`, shown in Figure 4.7), and might eventually wait (on a signaling mechanism) until the relevant parallel execution is finished. After the relevant parallel execution finishes, the producer can read the computed result from the piece of parallelism work. The producer (the thread that encountered the parallelism primitive) is considered to be the *parent*, and the consumer (worker) thread is considered to be the *child*. As shown in Figure 8.1, pieces of parallelism work go through the following five states: *unassigned*, *started*, *pending*, *resumed*, and *finished*.

A piece of work can also be classified as *active* (because it is associated with a worker thread that is consuming CPU cycles) or *inactive* (because it is unassociated with a worker thread, or its associated thread is blocked, waiting for some condition to occur). It is considered to be *active* if it is in either the *started* or *resumed* state and *inactive* when in the *unassigned*, *pending*, or *finished* state. In Figure 8.1, we color code the *active* states with light-green and the *inactive* states with light-red. We now explain each of the five states that a piece of parallelism work can go through.

1. *Unassigned* – The first classification refers to work not yet acquired by a worker thread. Until acquired by a worker thread, these pieces of work are stored in the global *parallelism work queue* (see Section 8.4.4 for implementation details of this queue). A piece of work remains *unassigned* until it is assigned a worker thread and that worker thread is assigned a CPU core upon which to execute. At this point, it transitions to the *started* state.

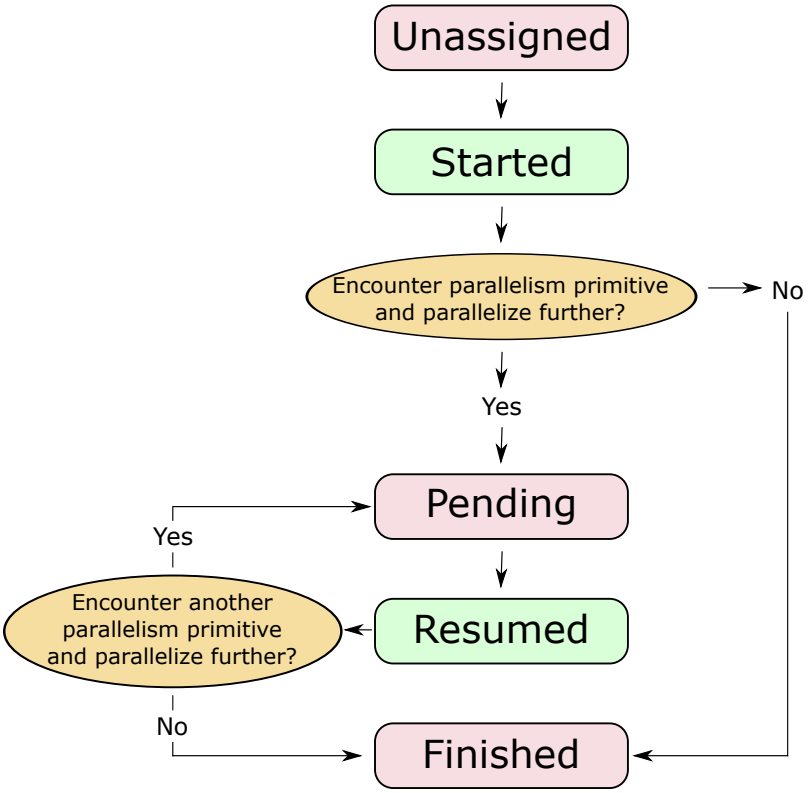

Figure 8.1: Life of a Piece of Parallelism Work

2. *Started* – This state describes the pieces of work that have been assigned to worker threads and have already started executing. These pieces of work have not yet encountered a parallelism primitive that would cause them to further parallelize computation. If the piece of work does not encounter any more parallelism primitives, the work finishes execution, its result is stored in the appropriate place, and it transitions to the *finished* state. If the piece of work does encounter a parallelism primitive and further parallelizes execution, the work transitions to the *pending* state.

3. *Pending* – This state refers to work that was *started* or *resumed* and encountered a parallelism primitive and decided to further parallelize execution. In this state, the piece of work is doing nothing and remains blocked until its child or children are finished executing. Once its children finish executing and its associated worker thread is assigned a CPU core, it transitions to the *resumed* state.

4. *Resumed* – Once a piece of parallelism work's child or children are finished executing, the work resumes execution. At this point, the piece of parallelism work is said to be in the *resumed* state. If the piece of work does not encounter any more parallelism primitives, the work finishes execution, its result is stored in the appropriate place, and it transitions to the *finished* state. If the piece of work does encounter another parallelism primitive and further parallelizes execution, the work returns to the *pending* state.

5. *Finished* – After the piece of parallelism work has saved its result in the appropriate place, the piece of work is removed from the work queue and becomes eligible for garbage collection by the Lisp. At this point, the returned value is only accessible by those with access to the ACL2 parallelism primitive that created the piece of parallelism work.

8.2 Using Threads to Implement Parallel Execution

We use threads to consume and execute pieces of parallelism work that are placed on the work queue by parallelism primitives. We call these threads *worker threads*. While a worker thread begins as a consumer of pieces of parallelism work, it may also become a producer if the piece of work it is executing encounters an ACL2 parallelism primitive.

Life of a Worker Thread When a worker thread is created, it goes through the following states. Figure 8.2 shows the path that a worker thread follows during its lifespan. Fundamental to the circular nature of a worker thread's lifespan is the idea that threads can be reused to process more than one piece of parallelism work. This is called *thread recycling* and is further explained in Section 8.4.2.

1. *Thread Start* – The starting state for any worker thread. After the host Lisp finishes initializing the thread, the thread transitions to the *idle* state.

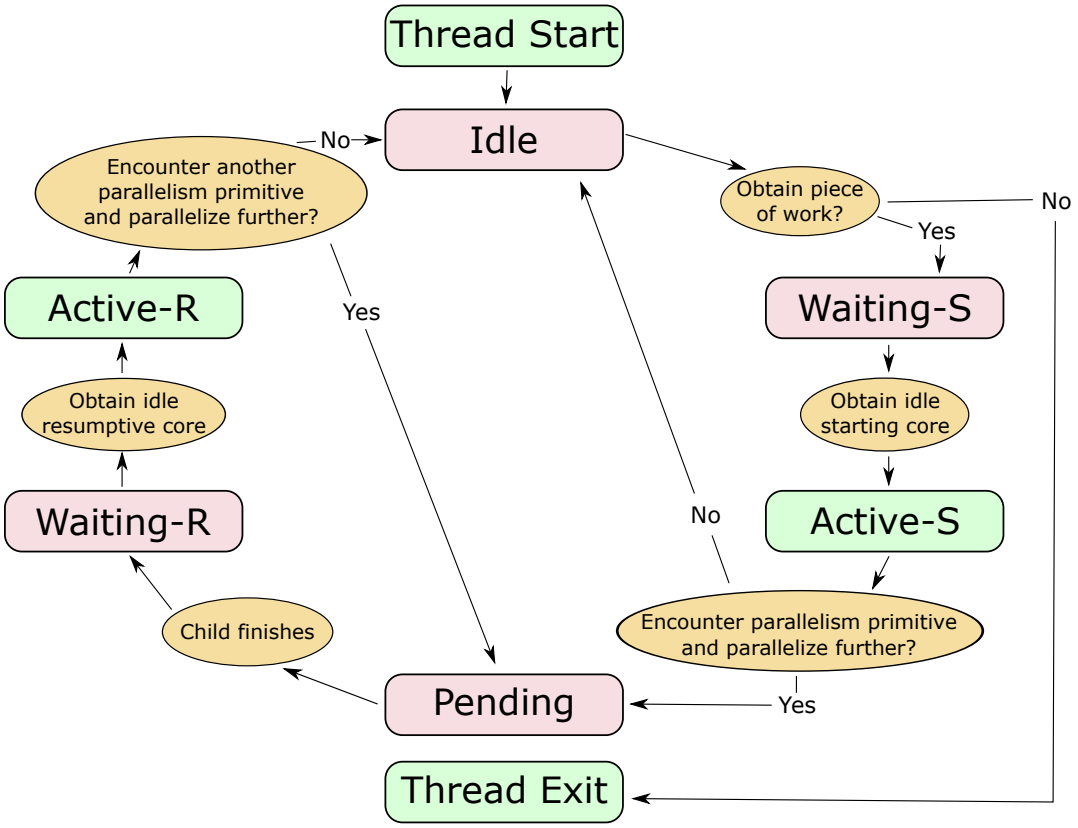

Figure 8.2: Life of a Worker Thread

2. *Idle* – The worker thread begins by waiting until there is a piece of parallelism work to consume. A worker thread will wait up to a fixed maximum amount of time for a piece of parallelism work. If a piece of parallelism work arrives before that amount of time elapses, the worker thread enters the *waiting-s* state. If a piece of parallelism work does not arrive before that amount of time elapses, the worker thread transitions to the *thread exit* state.

3. *Waiting-S* – After acquiring a piece of parallelism work to execute, the worker thread waits until there is an idle CPU core available for pieces of parallelism work that are just starting to execute. We only use the multi-threading primitives available to us in Lisp to manage CPU core resources (as opposed to trying to tell the OS how to schedule our threads). There is no timeout associated with this wait.

4. *Active-S* – Making it to (4) requires that the thread first made it through (2) and then also made it through (3). Thus, the thread has both a piece of parallelism work and is allocated a CPU core. At this point, the thread executes that piece of parallelism work. During this execution, one of two things happens.

   If the thread itself encounters another parallelism primitive and further parallelizes its execution, then the worker thread will do everything that ACL2 parallelism primitives do (adding the necessary piece[s] of parallelism work to the work queue, spawning more worker threads as needed,

and, once it needs the value from the parallelized computation, blocking until that value is available to be read). Once the worker thread blocks to read the value from the piece of parallelism work it generates, it is said to enter the *pending* state.

If the thread does not encounter another parallelism primitive, it stores the result of executing its assigned piece of parallelism work into the appropriate place and loops back to the *idle* state.

5. *Pending* (optional) – When a worker thread reaches the *pending* state, it has itself encountered a parallelism primitive and is now waiting for the results from the parallelized part of that primitive. There is no timeout associated with this wait, because, if the worker thread reaches this state, it needs the resulting value to continue execution.

6. *Waiting-R* (optional) – After the current worker thread is able to read the necessary values from parallelism primitives that it encounters, it attempts to resume execution. Since we still want to manage the number of threads that are allowed to execute at any one point in time, the thread must wait until there is an idle CPU core available for pieces of parallelism work that are resuming execution (see Section 8.4.3 for an explanation of what it means to wait for an idle CPU core to be available for pieces of parallelism work that are resuming execution). There is no timeout associated with this wait.

7. *Active-R* (optional) – After being allocated a CPU core, the thread re-

sumes execution and is once again considered to be *active-r*. If the worker thread does not encounter any additional parallelism primitives, the thread finishes the execution of the piece of parallelism work, stores the result of executing its assigned piece of parallelism work into the appropriate place, and loops back to the *idle* state. However, if the worker thread does encounter another parallelism primitive and parallelizes execution, the worker thread returns to the *pending* state.

8. *Thread Exit* – After performing some cleanup, the worker thread unwinds, is available for garbage collection, and is no longer associated with an OS thread.

8.3 Heuristics for Managing Parallelism Resources

The heuristics outlined in this section involve determining when to parallelize computation and imposing limits on the amount of parallelism so that the underlying runtime system does not become overwhelmed. In many parallelism examples, like the doubly recursive Fibonacci function, it is important to limit the amount of parallelism. Without such limits, the underlying Lisp and OS would no longer execute efficiently. Also, the overhead associated with computing the Fibonacci of small numbers (e.g., computing the fifth Fibonacci number) would far exceed the amount of time it would have taken to compute the result without using parallel execution.

Once we specifically targeted the ACL2 theorem proving process, the

mechanisms needed to obtain efficient execution changed. For example, the overhead for `spec-mv-let` is trivial when compared with the time it takes to prove any particular subgoal (see Section 8.3.1 for supporting details). As such, it is reasonable to parallelize the proof of any subgoal. This section delves into our heuristics that help us manage ACL2(p)'s parallel execution.

8.3.1 The Granularity of ACL2 Subgoals

In the ACL2 theorem prover, it turns out that accommodating granularity is a non-issue. This is because the overhead of parallelizing is typically much less than the duration it takes to prove any subgoal. As discussed in Section 4.3.2, the overhead of using `spec-mv-let` is between 33 and 45 microseconds. As such, one would hope that the number of subgoals that require around 45 microseconds or less to prove would be small. Indeed, this is the case. In Table 8.1, we categorize every subgoal in the ACL2 regression suite by the amount of time it takes to prove each subgoal (these timings were created on the same machine as those in Section 4.3.2). Of the 1,118,641 subgoals that make up the regression suite, 0.58% of them require less than 50 microseconds to complete their attempt at processing that specific subgoal, and 84.71% of them require more than 500 microseconds. Thus, we conclude that using `spec-mv-let` to parallelize the proofs of subgoals is a reasonable solution.

Range	Count of Subgoals	Percentage of Subgoals
1μ to 50μ	6435	0.58%
51μ to 100μ	34174	3.05%
101μ to 150μ	25641	2.29%
151μ to 200μ	16211	1.45%
201μ to 250μ	12565	1.12%
251μ to 300μ	13171	1.18%
301μ to 350μ	12374	1.11%
351μ to 400μ	13976	1.25%
401μ to 450μ	17119	1.53%
451μ to 500μ	19341	1.73%
$500+\mu$	947634	84.71%

Table 8.1: Number of Subgoals with Durations with the Given Time Range

8.3.2 Optimizing the Use of CPU Cores and Worker Threads

Our implementation seeks to optimize the use of two parallelism resources: CPU cores and threads. CPU cores are said to be in one of two states: *active* and *idle*. A CPU core is said to be *active* when a Lisp thread has been allocated to the core and is busy executing. Correspondingly, a CPU core is considered *idle* when the OS does not assign it a Lisp thread to execute. Since our implementation does not access the OS scheduler, it assumes that the current ACL2 session is the only application consuming significant CPU cycles. Given this assumption and a Lisp function that returns the total number of CPU cores, we can track the number of available CPU cores. As such, the implementation does not need to interact with the OS to make a thread runnable – threads that do not have our permission to run block on Lisp-level signaling mechanisms.

Figure 8.3 illustrates the relationships between pieces of work and their possible states, CPU cores, and worker threads.

Work State	unassigned	started	pending*	resumed*	finished
Allocated Core	no	yes	no	yes	no
Worker State	n/a	active-s	pending	active-r	n/a

*the pending and resumed states are not always entered.

Figure 8.3: Associations Between a Piece of Parallelism Work, CPU Cores, and Worker Threads

8.3.2.1 Limiting the Number of Active Worker Threads

The number of worker threads associated with *started* work is limited to the number of CPU cores in the system. Likewise, the number of worker threads associated with *resumed* work is also limited to the number of CPU cores in the system. Limiting the number of active threads in this way helps minimize context switching overhead [22]. We explain the implementation of this idea in Section 8.4.3.

We recognize that allowing more active threads in the system could allow the critical paths of proofs to begin executing sooner (an issue discussed in Section 7.3). If we assume that there is no penalty for context switching, this would allow them to finish sooner, potentially alleviating some of the performance issues discussed near the end of the introduction of Section 7.3. However, the penalty for context switching is non-trivial, and in preliminary experiments, it was better to limit the number of active threads as discussed in the prior paragraph. This being said, useful future work could include

increasing the number of allowed active threads and seeing how this affects performance for the theorems shown in tables 7.1, 7.2, and 7.4 and across the entire regression suite.

8.3.2.2 Keeping CPU Cores Busy

Whenever a worker thread acquires a piece of parallelism work from the *unassigned* section of the work queue, it immediately attempts to acquire an idle CPU core, and, once successful, the worker thread begins executing that piece of parallelism work. If there is no work in the *unassigned* section of the work queue, the worker thread will be idle until work is added. If opportunities for parallel execution were recently encountered but serial executions were performed because all CPU cores were busy, this idleness would be a wasted opportunity. To avoid this, the *unassigned* portion of the work queue is treated as a *buffer*, and we attempt to keep $3p$ pieces of work in it at all times. The number p represents the number of CPU cores (or in the case of a hyper-threaded machine, p represents the number of hardware threads), so that if all worker threads simultaneously finish executing their piece of parallelism work, they can acquire a new piece of work. It is probably fine for the buffer to be a little bit smaller or larger; we picked a threshold of $3p$, because it seems to work well for our application. One could easily size the buffer to contain p or $8p$ pieces of work, and, except in extreme examples, the change in performance would probably be unmeasurable.

Figure 8.4 shows the limits imposed upon a system with p CPU cores

and a limit l on the total number of pieces of work allowed to be in the parallelism system at once. In addition to keeping $3p$ pieces of parallelism work in the *unassigned* section, we also strive to keep p pieces of work in the *started* state, and we limit the number of pieces of work in the *resumed* state to p. This results in limiting the number of pieces of work in the *pending* state to approximately l - $5p$. The need for limit l is explained in the following section. The states in the figure that are light-green are consuming CPU core resources, while the light-red sections have no CPU core allocated to them.

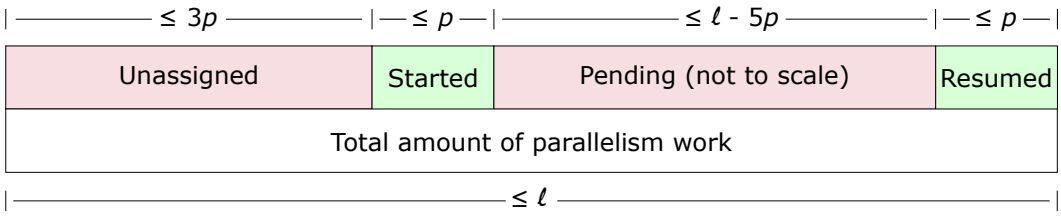

Figure 8.4: Limits for Each Category of Parallelism Work

8.3.2.3 Limiting Total Workload

Since the OS only supports a limited number of threads, restrictions must be imposed to ensure application stability. It is insufficient simply to set a limit on the number of worker threads spawned, because: (1) the stack of parents waiting on a deeply recursive nest of children can only unroll itself when the nest of children finishes executing (see the following paragraph labeled "Deeply Nested Trees of Parallelism Calls") and (2) any piece of work allowed into the system must eventually be assigned to a worker thread for processing. Further knowledge of the architecture of ACL2(p)'s parallelism

implementation is required to understand why we mention observation (2): Every parallelism primitive that produces pieces of parallelism work will usually spawn worker threads to execute this work. Therefore, before a parent can decide to add work, it must first determine whether the addition of work would require it to spawn more threads than are stable for the underlying runtime system. If the total count of already existing parallelism work is greater than a given limit (named l in Figure 8.4), the primitive opts for serial execution. See Section 6.2 for details concerning how users can manipulate this limit, which is set by default to 8,000 pieces of parallelism work.

Deeply Nested Trees of Parallelism Calls The following example demonstrates how execution can result in generating deeply nested parallelism calls that can require a large number of parent threads waiting on their child threads (who are in turn parents). Suppose there is a function that counts the leaves of a tree, as below:

```
(defun pcount (x)
  (declare (xargs :guard t))
  (if (atom x)
      1
    (pargs (binary-+ (pcount (car x))
                     (pcount (cdr x))))))
```

If this function is called on a heavily right-skewed tree, e.g., a list of length 100,000, then the computation may parallelize with every few `cdr` recursions. This creates a deeply nested call stack with potentially tens of thousands of `pargs` parents waiting on their `pcount` children. If the system

were allowed to create all of these threads, and if all of these threads were to become runnable at the same time, the Linux daemon Watchdog [35] could observe a high load average on the machine and cause the machine to reboot. So, we limit the total amount of parallelism work allowed in the system, which prevents the creation of these tens of thousands of threads, and thus, the system maintains stability.

While the above `pcount` example is extreme, we have encountered this problem when parallelizing ACL2 proofs. In an earlier implementation, processing a top-level goal that immediately case-splits into 800 subgoals required up to 800 threads when finishing the very last subgoal. This was because none of the threads could return until the last attempted subgoal was finished (whether it proved successfully or not is irrelevant). As such, there were many real examples where we needed to limit the total parallelism workload (e.g., theorem *measure-obligation-2* in book *coi/termination/assuming/complex.lisp*).

In response, we redesigned how we spawn parallel execution from within `waterfall1-lst`. In particular, the new parallel version of `waterfall1-lst` accepts a list of subgoals, splits the list into halves, and calls itself recursively with each half of the list (as opposed to calling itself recursively with the `cdr` of the list, which contains all but the first element of the list). This hierarchical approach requires more threads when starting a proof (it requires up to $2n$-1 threads), but it provides a more important benefit. Consider a goal, G, whose *nth* subgoal, H, is the only of its subgoals not yet completed. Then the proof for G will only have to wait on log n threads (instead of waiting

on n threads) while it waits for H to finish. We initially avoided making this change, because we wanted the code for the parallelized waterfall to match the non-parallelized version of the waterfall as closely as possible. However, increasing the stability of the underlying runtime system was important enough to warrant diverging from the non-parallel code base in this way. As a result, all but one of the books in the ACL2 regression suite can execute in parallel, from start to finish, without destabilizing the underlying runtime system (book *coi/termination/assuming/complex.lisp* still encounters this limit and serializes its execution for part of its proofs).

8.4 Optimizations

This section contains more detailed explanations of some of our optimizations. As the project progressed, the underlying Lisp implementations improved, and some of these optimizations are no longer necessary. Our work produced examples that motivated some of these improvements to CCL, SBCL, and LispWorks.

8.4.1 Semaphore Recycling

Semaphore recycling is an example of an optimization that was necessary for efficient execution at the beginning of our project but is no longer needed. In the past, the underlying Lisp (CCL) implementation did not free semaphores quickly enough for collection by the underlying OS. Thus, it was necessary to maintain a pool of previously allocated but then freed semaphores.

Without this pool, after allocating approximately 50,000 semaphores, with each new semaphore allocation, the Lisp would gradually begin to slow down and eventually become unresponsive.

Since then, garbage collection mechanisms for multi-threading primitives have improved. As such, we no longer need this semaphore allocation pool, but we mention it as an example of a practical engineering issue that we needed to overcome. Furthermore, the improvement of semaphore allocation is an example of how underlying Lisp implementations have improved in response to the performance challenges presented by our work.

8.4.2 Thread Recycling

Initial implementations spawned a fresh thread for each piece of parallelism work. The current implementation performs better than these original implementations, because, instead of letting threads expire after they finish a piece of parallelism work, they acquire or wait for a new piece of work from the work queue. The overhead associated with setting up the data structures necessary to recycle threads can be compared with the costs of creating new threads.

To do this comparison in CCL, consider the following scripts. The first script, shown in Figure 8.5, spawns two fresh threads to execute the arguments to the call `(binary-+ 3 4)`. The second script, shown in Figure 8.6, times the execution of `(pargs (binary-+ 3 4))`. Running the former script requires 0.562 seconds, and running the second script requires 0.039 seconds.

```
(defvar *x* 0)
(defvar *y* 0)
(defun parallelism-call ()
  (let* ((semaphore-to-signal (make-semaphore))
         (closure-1 (lambda ()
                      (prog1 (setf *x* 3)
                        (signal-semaphore
                          semaphore-to-signal))))
         (closure-2 (lambda ()
                      (prog1 (setf *y* 4)
                        (signal-semaphore
                          semaphore-to-signal))))
         (ignore1 (run-thread "closure-1 thread" closure-1))
         (ignore2 (run-thread "closure-2 thread" closure-2))
         (ignore3 (wait-on-semaphore semaphore-to-signal))
         (ignore4 (wait-on-semaphore semaphore-to-signal)))
    (declare (ignore ignore1 ignore2 ignore3 ignore4))
    (+ *x* *y*)))
(time (dotimes (i 1000)
        (assert (equal (parallelism-call) 7))))
```

Figure 8.5: Script to Determine the Cost of Computation When Spawning Fresh Threads

```
(time (dotimes (i 1000)
       (assert (equal (pargs (binary-+ 3 4)) 7))))
```

Figure 8.6: Script to Determine the Cost of Computation When Recycling Threads

These timing results suggest that it takes about 14.4 times longer to execute a parallelism call that spawns fresh threads instead of using `pargs`, which recycles threads. This script is run on *older-8-core-nht* (see Section 5.2.1 for the specifications of this machine).

Another way to measure the overhead of spawning threads is simply by timing how long it takes to spawn any number of threads to do a trivial computation. We determine how long it takes to spawn each thread with the test script shown in Figure 8.7. The script takes 39.0 seconds to run,

```
(defun do-nothing ())
(time (dotimes (i 100000) (run-thread "test thread"
                                      'do-nothing)))
```

Figure 8.7: Script to Determine the Cost of Spawning a Thread

so creating a thread requires 390 microseconds. Given executing a future requires less than 45 microseconds (see Section 4.3.1 for supporting details), we conclude that recycling threads is a useful optimization.

8.4.3 Resumptive Optimizations

Our explanation of the implementation has emphasized the case of a producer thread producing parallelism work (by encountering an ACL2 parallelism primitive) and having a consumer (worker thread) execute that piece of parallelism work. But what happens when the consumer itself encounters a parallelism primitive and becomes a producer? Once these consumer-producers' children finish, should they have a higher priority than the pieces of work still on the work queue? Consider the following simplified scenario. Suppose all the consumer-producer needs to do before finishing is apply a fast function like `binary-+` to the results of executing its arguments. Since, hopefully, the only work on the work queue is of reasonably large granularity, surely the application of `binary-+` would terminate sooner than executing a new piece of parallelism work. Also, if we allow the worker thread to finish a piece of parallelism work, an OS resource, a thread, becomes idle and available for executing more work. While there are scenarios that favor other priority schemes, a setup that allows the *resumed* piece of parallelism work to finish

with at least the same priority as threads in the *started* state (see Section 8.1) will likely free resources (threads) sooner and has been chosen for this implementation. Note that pieces of parallelism work that encounter multiple parallelism primitives may break our assumption that *resumed* pieces of parallelism work are likely to finish sooner than pieces of work that have just *started*. While this situation is a possibility, it does not occur in our application.

Implementing this scheme requires a second signaling mechanism, upon which only the worker threads that have generated child work themselves wait. When a waiting thread receives this mechanism's signal, it will claim an idle core in a more liberal fashion. Instead of waiting for the count of idle CPU cores to be positive, the thread will wait for the number of active worker threads to be less than or equal to twice the number of CPU cores in the system. For example, if there are 8 CPU cores and 8 active threads, then there can be up to 8 additional active threads that have become active through the resumptive mechanism. This could make a total of 16 active threads. As shown in Figure 8.2, after a resuming thread claims an idle core in this way, it is said to be active once again.

8.4.4 Work Queue Design

Our initial implementations of the work queue used a double-ended queue that was destructively modified any time a piece of parallelism work was added to or removed from the work queue. This required acquiring a shared lock during each addition or removal. Since these initial implementations,

atomic increments and decrements have been added to the underlying Lisps. Taking advantage of this, our latest implementation uses a shared array, and threads enqueuing or dequeuing a piece of parallelism work atomically increment shared global variables to retrieve the right slot for obtaining or storing a piece of parallelism work.

8.5 Debugging Parallel Performance

An interactive theorem proving system might have a mechanism for providing real-time feedback about which proof rules and heuristics are being applied. If someone parallelizing a theorem prover is looking for real time statistics on the parallelism system, they could consider repurposing such a mechanism to provide feedback on the parallelism system's performance.

We performed such a trick by reworking ACL2's DMR mechanism (described in the ACL2 Manual [1] in documentation topic "dmr") to provide real-time information about the number of active threads, amount of parallelism work in the system, and other debugging information that can be listed from calling our Lisp function `print-interesting-parallelism-variables`. We name this feature the "Parallelism Dashboard." Using this real-time information, we were able to discover that we had configured one of the limits for the system to be too low (see Section 8.3.2.3 for a discussion of this limit). Once we raised this limit, most of the proofs that were serializing computation were able instead to continue executing in parallel, and users experienced better overall performance. One can see an example of what the parallelism

dashboard presents in Figure 8.8.

```
Printing stats related to executing proofs in parallel.
Variable *IDLE-FUTURE-CORE-COUNT* is 32
Variable *IDLE-FUTURE-RESUMPTIVE-CORE-COUNT* is 40
Variable *IDLE-FUTURE-THREAD-COUNT* is 664
Variable *THREADS-WAITING-FOR-STARTING-CORE* is 0
Stat     NUMBER-OF-IDLE-THREADS-AND-THREADS-WAITING-FOR-
             A-STARTING-CORE is 664

Variable *UNASSIGNED-AND-ACTIVE-FUTURE-COUNT* is 8
Variable *UNASSIGNED-AND-ACTIVE-WORK-COUNT-LIMIT* is 160
Variable *TOTAL-FUTURE-COUNT* is 34
Stat     TOTAL-PARALLELISM-WORK-LIMIT is 10000

Stat     NUMBER-OF-ACTIVE-THREADS is 8
Stat     NUMBER-OF-THREADS-WAITING-ON-A-CHILD is 27

Variable *LAST-SLOT-TAKEN* is 3252
Variable *LAST-SLOT-SAVED* is 3252
Stat     FUTURE-QUEUE-LENGTH is 0
Stat     AVERAGE-FUTURE-QUEUE-SIZE is 770.65686

Variable *RESOURCE-BASED-PARALLELIZATIONS* is 0
Variable *RESOURCE-BASED-SERIALIZATIONS* is 0
Variable *RESOURCE-AND-TIMING-BASED-PARALLELIZATIONS* is 0
Variable *RESOURCE-AND-TIMING-BASED-SERIALIZATIONS* is 0
Variable *FUTURES-RESOURCES-AVAILABLE-COUNT* is 3250
Variable *FUTURES-RESOURCES-UNAVAILABLE-COUNT* is 0

Printing stats related to aborting futures.
Variable *ABORTED-FUTURES-TOTAL* is 0
Variable *ABORTED-FUTURES-VIA-THROW* is 0
Variable *ABORTED-FUTURES-VIA-FLAG* is 0
Variable *ALMOST-ABORTED-FUTURE-COUNT* is 0
```

Figure 8.8: Snapshot of Parallelism Dashboard Taken Near the End of Proving *[2b]* Using `Full` Waterfall Parallelism on *Modern-20-core-2ht*

Chapter 9

Development Procedure

This chapter explains some of our procedure for implementing ACL2(p). We break our procedure into the following steps. While some of these steps may appear on the surface to be simple, we emphasize that ACL2 is a large system with many lines of code (106,591 lines of actual code and a total of 275,457 lines in the source files of the current development copy of ACL2, as of this writing), and that making even small changes to the code-base can have far-reaching effects upon soundness and usability.

1. We created a version of our theorem prover mostly equivalent to the non-parallel version of the theorem prover. This version disabled the use of all sequential dependencies by using a macro system that caused a compile-time error any time it ran code that was inherently sequential. Note that being able to determine which code was inherently sequential was easier because ACL2 contains a mechanism that tracks changes to the global program state. This mechanism, unsurprisingly, is called *state*, and a description of it is available in Section 3.2.

2. Once we had a version of the theorem prover that was mostly thread-safe, we implemented the primitives and abstractions necessary to actually

execute the proof process in parallel. This required thinking about issues including but not limited to the following:

- Whether futures could be correctly incorporated into the theorem prover logic (we decided they could not be integrated without a very large amount of effort dedicated exclusively to this task).
- Whether we wanted to incorporate a mechanism that provides mutually exclusive execution into the logic (we did, see *deflock* in Section 4.1.3).
- Whether we needed to support speculative execution (we needed such support early on, but now we only need to support interrupts and aborts from the user).
- What types of support we needed from the underlying Lisp implementations (e.g., semaphore notification objects – see Section 4.1).

3. We then integrated the use of the parallelism primitive `spec-mv-let` (see Section 4.2.2.5) into the waterfall (see Section 3.1), causing the proof process to execute in parallel.

4. After actually parallelizing the execution of the theorem proving process, we used ACL2's regression suite to identify inherently sequential code that was not discoverable at compile-time (by either causing run-time errors in code that did not modify *state* but that we marked as inherently single-threaded anyway, or by witnessing the theorem prover breaking in

unexpected ways). Once we took note of the breakages, we disabled the use of each problematic book in the regression suite and continued on to find the next breakage.

One of the goals of our project and measurements of success is that we were able to reduce the number of books from the regression suite that fail under parallel execution from about 200 out of 3000 (7%) to 7 out of 3000 (0.23%). Having such a suite of broken proofs to fix was helpful in setting smaller goals for improving the implementation, and it gave us a method to track our progress.

5. We then started to think about how we could reincorporate the serial dependencies that we were forced to remove. The first dependency we wanted to restore was output.

   To restore output, we needed to think about what we really wanted to present to the theorem prover user. In this step, we were fortunate, because, as discussed in sections 3.3 and 7.2.1, the ACL2 maintainers had already identified a useful subset of the output. As such, our goal became to mimic that output. Also, since there was so much less of this output, we were able to ignore output ordering issues and guarantee atomicity of printing by just using a mutual exclusion mechanism (locks). We were successful in this endeavor of providing useful output.

   It was at this point that users began receiving feedback sooner than they would have received it if they were using the non-parallel version of the

theorem prover.

6. Now that we had a product that was useful, we needed to continue the process of iterating over the set of removed features and fixing those features. We list the details for some of the features below.

   Note that it would have been nearly impossible to reincorporate these features in phases if we had not developed (and continued to refine) a macro system that allowed us to define both a serial version of a function and the parallel version of the same function, without having to duplicate most of the function body's code. This feature eventually evolved into our "defun@par" macro system. This macro system allows us to simultaneously define: (1) one version of a function that is trusted by the ACL2 authors for use in the non-parallel version of ACL2 and also for use in ACL2(p) when waterfall parallelism is disabled and (2) another version of the function that only exists in ACL2(p) and is only used when executing the waterfall in parallel. If a function is unable to be defined using "defun@par" (or if we needed a macro), it is still possible to define what we need in a way that is consistent with this scheme and gives us the properties of (1) and (2), described above. Any large software engineering project written in a language with a powerful macro system that requires both (1) and (2) could consider an approach similar to ours, which is documented in the ACL2 distribution under the definition of constant `*@par-mappings*` inside file *axioms.lisp* [1].

Some of the features that we needed to reincorporate include, but are not limited to, the following:

- We reworked the mechanism that ACL2 uses to store and access the theory that should be used to prove any particular subgoal: the *enabled-structure*.
- We removed the need to modify the program state from within the mechanism that translates user-level terms into their internal representation. This allowed us to reincorporate the use of the ACL2 *translator* from within the main proof process.
- We reworked the theorem proving process to call a version of the *evaluator* that does not modify the program state.
- We needed to *provide feedback to users when they tried to give hints to the theorem prover that were inherently sequential*. We also provide mechanisms that let users accept the risk of using their single-threaded hints and continue anyway. The affected hint mechanisms include *computed hints*, *custom keyword hints*, and *override hints*. The user-level discussion of this is available in Section 6.3.
- We needed to improve the user experience with regards to *clause-processors*. Specifically, we now cause a clean error whenever users attempt to use a clause-processor that modifies *state* while executing the waterfall in parallel. We also provide mechanisms that let

users accept the risk of using their single-threaded clause-processor and continuing anyway.

7. We created a mechanism for dynamically monitoring the underlying parallel execution system. This allows us to debug the performance of proofs that we expected to experience non-trivial amounts of speedup but that were not performing well. This debugging mechanism is described in Section 8.5.

8. Since we had provided both non-parallel and parallel versions of the waterfall in ACL2(p), we created a mechanism for users to dynamically switch between them. A user-level view of this mechanism is available in Section 6.1.

9. We ported our implementation to an additional platform, LispWorks. This gave us another Lisp implementation on which to debug problems. It also helped us ascertain whether bugs were Lisp implementation bugs or caused by problems in ACL2(p)'s code.

10. Finally, we quantified the improved performance for every theorem in the regression suite. We accomplished this by comparing the time required to prove the theorem using parallel execution with the time required to prove it serially. Our method for categorizing proofs based on the way that they do or do not benefit from parallel execution is articulated in Chapter 7. This chapter also provides example proofs for each of these categories.

Chapter 10

Future Work and Conclusion

In this chapter we discuss our vision and future work for using parallel execution to improve interactive theorem provers. We also discuss the future work specifically related to ACL2(p). We then conclude with a summary of our work.

10.1 Future Work for Interactive Theorem Proving

As parallel execution becomes more prevalent throughout interactive theorem prover systems, we will find that the expectations that users have from the tool change. As an example, currently, trained theorem prover users will often think of lemmas that prevent the theorem prover from exploding into many case-splits. However, now that parallelism is available to rapidly process such case-splits, there is less need for users to invest time in such thought. Instead, users can submit the theorem, and the many available CPU cores can do the work. Additionally, while past users may have needed to wait many seconds for relevant feedback, with sixty-four available CPU cores, the feedback could now arrive nearly instantaneously. So, while in the past, users were encouraged to invest energy in streamlining a proof, we might now

actually encourage them to save their time and energy and incur many case splits, letting the theorem prover perform the difficult work.

The penalty for distracting users with many different proof attempt strategies has been high – such strategies could significantly delay the time required to arrive at the strategy that worked. However, with the arrival of systems with an arbitrarily large number of CPU cores, the penalty of such delays becomes a non-issue. Therefore, the theorem proving community will have more incentive in the future to present strategies to the theorem prover that are less likely to work, so that the theorem prover can try them in the background of the main proof attempt. We describe an ACL2-specific mechanism called *or-hints* in the next section that is a single-threaded implementation example of this idea.

As described in Section 6.1 (under the heading "Resource and Timing Based"), having a mode of parallel execution in *interactive* theorem provers that considers information from previous proof attempts of a particular theorem could improve the performance of subsequent proof attempts. Our first attempt at such a mode is described in that section, along with details concerning how to better implement such a mode. The idea for a future version of this mode is that one can use timing information from previous proof attempts to prioritize, in future proof attempts, the subgoals along the critical path of the proof. Using timing information from failed proof attempts in this way would further sidestep (using parallel execution already partly sidesteps) the need to develop heuristics that predict the time required to complete any

particular subgoal's proof attempt.

10.2 Future Work for ACL2(p)

Another place where we could install parallel execution is in ACL2's implementation of *or-hints*, a mechanism for users to suggest two or more proof strategies to try, instead of suggesting just one proof strategy (see topic "hints" in the ACL2 Manual [1]). ACL2 currently implements or-hints by attempting each set of user-provided strategies, one after another, within a single thread. Clearly it would be more advantageous to execute these hints in parallel on separate CPU cores. Given that or-hints are implemented at a higher functional level within the ACL2 code (and because the use of or-hints is not dominant throughout the ACL2 regression suite), we did not modify ACL2(p) to implement this idea, choosing instead to focus on more immediate opportunities for parallel execution. However, if we were to implement such an idea, we would likely see another paradigm shift as users attempted to use different proof strategies in parallel. Indeed, we already see users attempt to use different proof strategies without parallel execution; e.g., instead of manually trying each version of the ACL2 arithmetic libraries when reasoning about arithmetic operations, there is a book (*make-event/proof-by-arith.lisp*) that can be used to attempt the use of each library. If any of the arithmetic libraries succeeds in completing the proof, this book lets the user determine which library succeeded. In the future, the attempted use of each of these arithmetic libraries could be performed in parallel and users could increase

their reliance upon such techniques.

There are also many ACL2-specific implementation issues that we leave as future work. In general, we leave the reader to examine the documentation topic "unsupported-waterfall-parallelism-features" in the ACL2 manual [1]. However, we mention a few issues here. To begin with, after further hardening of the SMP implementation of ACL2(p), it may be productive to distribute proof obligations across a network of computers (mentioned in Section 2.3). Another possibility for future work includes building `plet` and `pargs` on top of the futures interface instead of having them connect directly with the multi-threading primitives like locks and semaphores (mentioned in the introduction to Chapter 4). Some possible avenues for further tuning of the system involve adjusting our limit on the number of threads allowed to be *active* (discussed in Section 8.3.2.1), adjusting our limit on the size of the *unassigned* section of the work queue (discussed under the heading "Case Study: Theorem *Ub-g-chain-=-g-chain-skolem-f*" in Section 7.2.4), and developing more ways of managing the garbage collection threshold (discussed in Section 7.2.4). Finally, we highlight one feature that we did not implement but that could be particularly interesting to finish for ACL2(p): arrays. Arrays are by nature single-threaded, but ACL2 has an implementation of arrays that already detects when an array has been modified in an "unauthorized way". This "unauthorized way" could be generalized to include modifications from "unauthorized" threads. Also, the association lists that implement arrays when "unauthorized changes" occur could be made thread-local, and we could attempt to find a way to merge

these changes back into the global array once the thread finishes a piece of parallelism work. This is an example of how ACL2's efforts to provide a platform for efficient execution while also maintaining soundness makes ACL2 a good platform for exploring how to provide thread-safe versions of inherently single-threaded programs.

10.3 Summary

By introducing parallel execution into ACL2's main proof process, the waterfall, we have lessened the delay between when users submit a conjecture to the prover and when they receive feedback from the prover that helps them debug their proof. This early feedback is achieved by (1) reducing the overall time required to attempt a proof and (2) presenting output such that coherent feedback is given sooner that helps users more quickly investigate failed proof attempts.

We achieved the above goal by: (1) implementing and incorporating the mechanisms necessary to execute ACL2 programs in parallel, (2) modifying ACL2 so that it can continue to be used interactively even though we execute its main proof process in parallel, and (3) categorizing a broad set of ACL2 theorems, known as the regression suite, based on their amenability to parallel execution.

Bibliography

[1] ACL2. *ACL2 Documentation*, January 2012. http://www.cs.utexas.edu/-users/moore/acl2/current/acl2-doc-index.html.

[2] Gul Agha. An overview of actor languages. *SIGPLAN Notices*, 21(10):58–67, 1986.

[3] Hassan Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, Cambridge, MA, USA, 1991.

[4] Argonne National Laboratory. *The Message Passing Interface (MPI) Standard*, March 2010. http://www.mcs.anl.gov/index.php.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[6] Maria Paola Bonacina and William McCune. Distributed theorem proving by peers. In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 841–845, London, UK, 1994. Springer-Verlag.

[7] Bordeaux Threads. *Bordeaux Threads API Documentation*, March 2010. http://trac.common-lisp.net/bordeaux-threads/wiki/ApiDocumentation.

[8] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam K. Srivas and Albert John Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 275–293. Springer-Verlag, 1996.

[9] Clozure Associates. *Clozure CL Documentation*, March 2010. `http://ccl.clozure.com/manual`.

[10] Chance Elliott, Vipin Vijayakumar, Wesley Zink, and Richard Hansen. National Instruments LabVIEW: A programming environment for laboratory automation and measurement. *Journal of Laboratory Automation*, 12(1):17–24, February 2007.

[11] Matthew Fluet, Lars Bergstrom, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Programming in Manticore, a heterogenous parallel functional language. In Zoltn Horvth, Rinus Plasmeijer, and Viktria Zsk, editors, *Central European Functional Programming School*, volume 6299 of *Lecture Notes in Computer Science*, pages 94–145. Springer Berlin / Heidelberg, 2010.

[12] Richard P. Gabriel and John McCarthy. Queue-based multi-processing Lisp. In *Conference on Lisp and Functional Programming*, pages 25–44, 1984.

[13] Emden R. Gansner and John H. Reppy. A multi-threaded higher-order user interface toolkit. In *User Interface Software, Bass and Dewan*

(Eds.), volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

[14] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a microprocessor. In *Conference on Lisp and Functional Programming*, pages 9–17, 1984.

[15] Robert H. Halstead, Jr. New ideas in parallel Lisp: Language design, implementation, and programming tools. In *Parallel Lisp: Languages and Systems*, pages 2–57, 1989.

[16] Kecheng Hao, Fei Xie, Sandip Ray, and Jin Yang. Optimizing equivalence checking for behavioral synthesis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1500–1505, March 2010.

[17] Williams Ludwell Harrison. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3):179–396, 1989.

[18] Williams Ludwell Harrison and Zahira Ammarguellat. A comparison of automatic versus manual parallelization of the boyer-moore theorem prover. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 307–330, London, UK, 1990. Pitman Publishing.

[19] Haskell.org. *Glasgow Haskell Compilation System User's Guide*, July 2012. http://www.haskell.org/ghc/docs/7.4.2/html/users_guide/lang-

-parallel.html.

[20] IEEE. Standard for information technology - portable operating system interface (POSIX). Technical report, 2004.

[21] Intel Inc. Intel turbo boost technology - on demand processor performance. On the Web, February 2012. http://www.intel.com/content/-www/us/en/architecture-and-technology/turbo-boost/turbo-boost--technology.html.

[22] SL Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[23] Deepak Kapur and Mark T. Vandevoorde. DLP: a paradigm for parallel interactive theorem proving, 1996.

[24] Matt Kaufmann and J Strother Moore. Some key research problems in automated theorem proving for hardware and software verification. *Spanish Royal Academy of Science (RACSAM)*, 98(1):181–195, 2004.

[25] Matt Kaufmann and J Strother Moore. An ACL2 tutorial. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 17–21, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] Matt Kaufmann and Matt Wilding. A parallel version of the boyer-moore prover. Technical Report 39, Computational Logic, Inc., February 1989.

[27] David Kitchin, Adrian Quark, William R. Cook, and Jayadev Misra. The Orc programming language. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *Proceedings of FMOODS/FORTE 2009*, volume 5522 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2009.

[28] Lawrence Berkeley National Laboratory and UC Berkeley. *Berkeley UPC - Unified Parallel C*, March 2010. http://upc.lbl.gov.

[29] Reinhold Letz, Johann Schumann, Stefan Bayerl, and Wolfgang Bibel. SETHO: A high-performance theorem prover. In *Journal of Automated Reasoning*, volume 8(2), pages 183–212, 1992.

[30] LispWorks. *LispWorks Reference Manual*, January 2012. http://www.-lispworks.com/documentation/lw61/LW/html/lw.htm.

[31] Frédéric Loulergue, Wadoud Bousdira, Frédéric Gava, Louis Gesbert, Gaétan Hains, Guillaume Petiot, and Julien Tesso. *Bulk Synchronous Parallel ML 0.5 Reference Manual*. University of Orléans, October 2010. http://citeseer.ist.psu.edu/article/wolf97cooperative.html.

[32] Frédéric Loulergue, Frédéric Gava, Myrto Arapinis, and Frédéric Dabrowski. Semantics and implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, 5(3):182–199, 2004.

[33] David C. J. Matthews and Makarius Wenzel. Efficient parallel programming in poly/ML and isabelle/ML. In *DAMP '10: Proceedings of the 5th*

ACM SIGPLAN workshop on Declarative aspects of multicore programming, pages 53–62, New York, NY, USA, 2010. ACM.

[34] José Meseguer and Timothy C. Winkler. Parallel programmming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, pages 253–293, London, UK, 1992. Springer-Verlag.

[35] Michael Meskes. Watchdog: The Linux software daemon. *Linux Journal*, 1997(34es), February 1997.

[36] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[37] MIT Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, March 2010. http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf.

[38] J Strother Moore and George Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems*, 24:193–216, May 2002.

[39] Roderick Moten. Exploiting parallelism in interactive theorem provers. In *Theorem Proving in Higher Order Logics*, pages 315–330, 1998.

[40] Charles J. Northrup. *Programming with UNIX threads*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[41] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in*

Artificial Intelligence, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[42] Ken Pitman. *The Common Lisp Hyperspec*. Harlequin group ltd, http://www.harlequin.com/education/books/HyperSpec/FrontMatter/--index.html, 1996.

[43] David L. Rager. Adding parallelism capabilities in ACL2. In *ACL2 '06: Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 90–94, New York, New York, USA, 2006. ACM.

[44] David L. Rager. Implementing a parallelism library for ACL2 in modern day Lisps. Master's thesis, The University of Texas at Austin, 2008.

[45] David L. Rager and Warren A. Hunt, Jr. Implementing a parallelism library for a functional subset of Lisp. In *Proceedings of the 2009 International Lisp Conference*, pages 18–30, Sterling, Virginia, USA, 2009. Association of Lisp Users.

[46] David L. Rager, Warren A. Hunt, Jr., and Matt Kaufmann. A futures library and parallelism abstractions for a functional subset of Lisp. In *Proceedings of the 4th European Lisp Symposium*, March 2011.

[47] Sandip Ray and Jayanta Bhadra. A mechanized refinement framework for analysis of custom memories. In Jason Baumgartner and Mary Sheeran,

editors, *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2007)*, pages 239–242, Austin, TX, November 2007. IEEE Computer Society.

[48] Sandip Ray, Jayanta Bhadra, Thomas Portlock, and Ronald Syzdek. Modeling and verification of industrial flash memories. In *Quality Electronic Design (ISQED), 2010 11th International Symposium on Quality Electronic Design*, pages 705–712, March 2010.

[49] Sandip Ray, Kecheng Hao, Fei Xie, and Jin Yang. Formal Verification for High-Assurance Behavioral Synthesis. In Zhiming Liu and Anders P. Ravn, editors, *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA 2009)*, volume 5799 of *LNCS*, pages 337–351, Macao SAR, China, October 2009. Springer.

[50] Sandip Ray and Warren A. Hunt, Jr. Mechanized Certification of Secure Hardware Designs. In Magdy S. Abadir, Li-C. Wang, and Jayanta Bhadra, editors, *Proceedings of the 8th International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2007)*, pages 25–32, Austin, TX, December 2007. IEEE Computer Society.

[51] Sandip Ray and Warren A. Hunt, Jr. Connecting pre-silicon and post-silicon verification. In *Formal Methods in Computer-Aided Design, 2009*, pages 160–163, November 2009.

[52] John Reppy, Claudio Russo, and Yingqi Xiao. Parallel concurrent ml. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 257–268, September 2009.

[53] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[54] David Russinoff, Matt Kaufmann, Eric Smith, and Robert Sumners. Formal verification of floating-point RTL at AMD using the ACL2 theorem prover. In Simonov Nikolai, editor, *Proceedings of the 17th IMACS World Congrress on Scientific Computation, Applied Mathematics and Simulation*, Paris, France, 2005.

[55] SBCL. *SBCL Manual*, March 2010. http://www.sbcl.org/manual/.

[56] Johann Schumann. SicoTHEO: Simple competitive parallel theroem provers. In *Conference on Automated Deduction*, pages 240–244, 1996.

[57] Johann Schumann. *Automated theorem proving in software engineering*. Springer, 2001.

[58] Johann Schumann and Reinhold Letz. PARTHEO: A high-performance parallel theorem prover. In *Conference on Automated Deduction*, pages 40–56, 1990.

[59] SRI International. *PVS Specification and Verification System*, July 2012. http://pvs.csl.sri.com/.

[60] University of Cambridge. *HOL4 Kananaskis 5*, March 2010. http://hol.-sourceforge.net/.

[61] David H. D. Warren. An abstract prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, October 1983.

[62] Makarius Wenzel. Parallel proof checking in Isabelle/Isar. In Gabriel Dos Reis and Laurent Théry, editors, *ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS)*. ACM Digital library, August 2009.

[63] Andreas Wolf and Marc Fuchs. Cooperative parallel automated theorem proving. Technical report, Munich University of Technology, 1997.

[64] Cliff Young, Lakshman Y.N., Tom Szymanski, John Reppy, David Presotto, Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse. Protium: An infrastructure for partitioned applications. In *Proceedings of the Eighth Workshop on Hot Operating Systems (HotOS-VIII)*, January 2001.

[65] C. K. Yuen. *Parallel Lisp Systems: A Study of Languages and Architectures*. Chapman & Hall, Ltd., London, UK, UK, 1992.